МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Дрогобицький державний педагогічний університет імені Івана Франка

Ірина ШАКЛЕІНА, Андрій ПОПОВИЧ

Об’єктно-орієнтоване програмування:

методичні матеріали до самостійної роботи

для підготовки фахівців першого (бакалаврського)
рівня вищої освіти
за напрямом підготовки F3 Комп’ютерні науки

Дрогобич
2025

УДК 004(07)
ББК 32.973.202

Рекомендовано до друку вченою радою
Дрогобицького державного педагогічного університету імені Івана Франка
(протокол №19 від 23 .10. 2025 р.)

Р е ц е н з е н т и :

Ришковець Юрій Володимирович — кандидат техн. наук,. доцент кафедри інформаційних систем і мереж національного університету «Львівська політехніка»

Оршанський Леонід Володимирович — доктор пед. наук,. професор, завідувач кафедри технологічної та професійної освіти Дрогобицького державного педагогічного університету імені Івана Франка

Шаклеіна І.О., Попович А.В.

Об'єктно-орієнтоване програмування: методичні матеріали до самостійної роботи [для підготовки фахівців першого (бакалаврського) рівня вищої освіти напряму підготовки F3 «Комп'ютерні науки»] / І.О. Шаклеіна, А.В. Попович – Дрогобич : Видавничий відділ Дрогобицького державного педагогічного університету імені Івана Франка, 2025. – 92 с.

Навчально-методичні матеріали до самостійної роботи розроблено відповідно до програми навчальної дисципліни «Об'єктно-орієнтоване програмування» для підготовки фахівців першого (бакалаврського) рівня вищої освіти напрямів підготовки F3 Комп'ютерні науки. Матеріали до самостійної роботи містять теми лекцій та теми лабораторних робіт з методичними рекомендаціями для підготовки до них, питання для самостійного опрацювання і завдання для самоперевірки за даним матеріалом, детальний опис засобів та форм контролю знань; методичні поради щодо виконання індивідуального завдання, зразки практичних та тестових завдань, винесених на контрольні роботи, зразок екзаменаційного білету, питання до іспиту, глосарій термів до курсу та список навчально-методичної літератури для самопідготовки з дисципліни "Об'єктно-орієнтоване програмування"

ЗМІСТ

ПЕРЕДМОВА

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням складності програмних систем та підвищеними вимогами до їх якості, надійності й масштабованості. У цьому контексті об’єктно-орієнтоване програмування (ООП) посідає провідне місце серед методологій створення програмного забезпечення. Саме принципи ООП – інкапсуляція, успадкування та поліморфізм – дають змогу проектувати та розробляти системи, які легко супроводжуються, розширюються та відповідають сучасним потребам ринку праці. Тому вивчення ООП є необхідною складовою підготовки майбутніх фахівців з інформаційних технологій.

Посібник для самостійної роботи з дисципліни «Об’єктно-орієнтоване програмування» укладений для допомоги в організації самостійної роботи студентів, які вивчають основи об'єктно-орієнтованого програмування та прагнуть отримати ґрунтовні теоретичні знання в поєднанні з практичними навичками. Самостійна робота студентів являє собою навчальну діяльність, що ґрунтується на власній ініціативі студентів при опосередкованому керівництві з боку викладачів. Даний вид роботи забезпечує засвоєння професійних компетенцій та сприяє розвитку таких якостей особистості, як ініціативність та автономність у навчанні.

Самостійна робота над навчальним матеріалом спрямована на формування універсальних навичок автономної діяльності у різних сферах: освітній, дослідницькій та майбутній професійній практиці. Через самостійну роботу студенти розвивають здатність брати відповідальність за власні рішення, творчо підходити до вирішення складних завдань, знаходити оптимальні шляхи подолання труднощів та ефективно діяти у нестандартних ситуаціях.

Матеріал посібника структуровано таким чином, щоб забезпечити послідовне засвоєння концепцій ООП – від базових понять до складних

архітектурних рішень та патернів проєктування. Посібник побудовано за принципом поступового ускладнення матеріалу та інтеграції теоретичних знань з практичною діяльністю. Кожен розділ логічно пов'язаний з попередніми та наступними, створюючи цілісну систему знань.

Лекційний матеріал охоплює фундаментальні концепції ООП, включаючи інкапсуляцію, наслідування, поліморфізм та абстракцію. Особлива увага приділяється розумінню принципів SOLID, які є основою якісного об'єктно-орієнтованого дизайну. Теоретичні положення супроводжуються практичними прикладами та ілюстраціями, що допомагають студентам краще зрозуміти абстрактні концепції. Викладення матеріалу відбувається з урахуванням сучасних тенденцій розвитку програмних технологій та потреб індустрії розробки програмного забезпечення.

Лабораторні заняття розроблено з метою закріплення теоретичних знань через практичну роботу. Завдання поступово ускладнюються – від створення простих класів до реалізації складних ієрархій успадкування та впровадження патернів проєктування. Кожна лабораторна робота містить чіткі інструкції, приклади виконання та критерії оцінювання, що дозволяє студентам самостійно контролювати якість своєї роботи. Практичні завдання спрямовані на формування навичок аналізу предметної області, проєктування програмних рішень та їх реалізації з використанням сучасних інструментів розробки.

Особливу увагу в посібнику приділено методичним рекомендаціям щодо організації самостійної роботи. Студентам запропоновано план опрацьовування теоретичного матеріалу з можливістю перевірки рівня розуміння основних положень, виконувати запропоновані практичні завдання, формулювати власні приклади для перевірки засвоєних понять. Матеріал для самостійного опрацювання розширює межі аудиторних занять та включає сучасні тенденції в ООП, такі як використання фреймворків, патерни проєктування GoF та GRASP, принципи чистого коду та рефакторингу. Цей розділ особливо важливий для

студентів, які прагнуть поглибити свої знання та підготуватися до професійної діяльності. Самостійна робота сприяє розвитку навичок дослідницької діяльності, критичного мислення та здатності до безперервного навчання, що є ключовими компетенціями сучасного ІТ-фахівця.

При укладанні посібника використано компетентнісний підхід, який передбачає формування не лише знань, але й практичних умінь та навичок. Особлива увага приділена розвитку алгоритмічного мислення, здатності до аналізу та синтезу при проєктуванні програмних систем. Матеріал викладено з урахуванням сучасних стандартів програмування та індустріальних практик. Приклади коду відповідають актуальним версіям мов програмування та демонструють не лише синтаксичні особливості, але й стилістичні рекомендації професійної розробки.

Посібник містить комплексну систему оцінювання знань студентів, яка включає різні форми контролю. Поточний контроль здійснюється через виконання лабораторних робіт, контрольних завдань та тестів, що дозволяє відслідковувати прогрес засвоєння матеріалу на кожному етапі навчання. Така система забезпечує своєчасне виявлення прогалин у знаннях та можливість їх корекції до підсумкового контролю.

Індивідуальні завдання мають на меті розроблення програмного застосунку з використання технологій ООП та командного підходу до реалізації ІТ-проєктів. Ці завдання мотивують студентів до поглибленого вивчення окремих аспектів ООП та формують навички дослідницької роботи. Контрольна робота дозволяє оцінити рівень засвоєння ключових концепцій та здатність до їх практичного застосування в типових ситуаціях.

Глосарій термінів та рекомендована література дозволяють студентам поглибити свої знання та ознайомитися з додатковими джерелами інформації. Література підібрана таким чином, щоб охопити як класичні праці з ООП, так і сучасні публікації, що відображають актуальні тенденції в галузі.

Для ефективного засвоєння матеріалу рекомендується дотримуватися послідовності вивчення розділів та не переходити до наступної теми без повного розуміння попередньої. Особливо важливо приділяти достатню увагу практичним завданням, оскільки саме через програмування формується справжнє розуміння об'єктно-орієнтованих принципів. Студентам варто активно використовувати сучасні інтегровані середовища розробки, які надають потужні інструменти для написання, налагодження та рефакторингу коду.

Рекомендується також ознайомлюватися з актуальною документацією мов програмування та фреймворків, що сприятиме формуванню навичок самостійного навчання. Важливим аспектом є участь у професійних спільнотах програмістів, вивчення відкритого коду успішних проєктів та слідування найкращим практикам індустрії розробки програмного забезпечення.

Об'єктно-орієнтоване програмування є не лише технічною дисципліною, але й способом мислення, який формує підходи до вирішення складних задач через декомпозицію, абстракцію та модульність. Засвоєння принципів ООП відкриває перед студентами широкі можливості для професійного зростання в галузі розробки програмного забезпечення. Сучасний ринок праці потребує фахівців, які володіють не лише технічними навичками, але й здатні мислити системно, працювати в команді та адаптуватися до швидко змінюваних технологій.

Даний посібник створено з метою забезпечення студентів всіма необхідними інструментами для успішного вивчення ООП та подальшого застосування отриманих знань у професійній діяльності. Автори висловлюють сподівання, що посібник стане в нагоді студентам у їхній освітній діяльності, сприятиме підвищенню рівня їхньої самостійності та підготує ґрунтовну базу для подальшого поглибленого вивчення сучасних мов програмування та технологій програмної інженерії.

МАТЕРІАЛ, ВИНЕСЕНИЙ НА ЛЕКЦІЙНІ ЗАНЯТТЯ

I СЕМЕСТР

Тема 1. Основні поняття ООП. Класи і об'єкти.

1. Вступ до об'єктно-орієнтованого програмування (ООП):

- ООП як сучасна методологія програмування.
- Основні поняття: об'єкт, клас, атрибути, методи.
- Абстракція даних і роль ООП у спрощенні складних програмних систем.

2. Базові принципи ООП:

- Інкапсуляція (приховування деталей реалізації).
- Успадкування (створення ієрархій класів).
- Поліморфізм (єдиний інтерфейс – різна реалізація).
- Приклади застосування у програмних системах.

3. Об'єктно-орієнтований аналіз та моделювання:

- Декомпозиція складних об'єктів.
- Побудова інформаційних моделей.
- Основи UML (клас-діаграми, діаграми об'єктів).

4. Класи і об'єкти в C++:

- Оголошення класів. Відкриті (public), закриті (private) та захищені (protected) члени класу.
- Об'єкти, масиви об'єктів.
- Вказівники на об'єкти, специфікатор this.
- Динамічне створення і знищення об'єктів (new, delete).
- Посилання в C++: передача об'єктів у функції, повернення об'єктів із функцій.
- Вбудовані (inline) функції в оголошенні класу.

5. Розширені можливості класів у C++:

- Статичні методи і змінні: призначення, особливості використання.
- Константні методи.

- Дружні функції (friend).

 ***Запитання для самоконтролю за темою 1***

1. Що таке об’єктно-орієнтоване програмування і які основні переваги воно має порівняно з процедурним підходом?
2. Що таке клас і об’єкт? Наведіть приклади з реального життя.
3. У чому полягає суть абстракції даних?
4. Поясніть принципи інкапсуляції, успадкування та поліморфізму. Наведіть приклади кожного.
5. Що таке об’єктно-орієнтований аналіз і як він пов’язаний з UML?
6. Які є специфікатори доступу у класах C++ і чим вони відрізняються?
7. Як створюються та використовуються масиви об’єктів у C++?
8. Для чого потрібні оператори new і delete у C++?
9. Що таке статичні та константні методи класу? У чому їх особливості?
10. Що таке дружні функції і коли доцільно їх використовувати?

Тема 2. Конструктори та деструктори. Вкладені класи. Композиція класів

1. Конструктори класів у C++

- Поняття конструктора. Призначення конструктора для ініціалізації об’єктів.
- Особливості іменування та виклику конструктора.
- Види конструкторів: конструктор без параметрів, конструктор з параметрами, конструктор копіювання.
- Використання ініціалізаторів у списку конструктора.
- Виклик одного конструктора з іншого (delegating constructors у сучасному C++).

2. Деструктори:

- Призначення деструктора.

- Автоматичний виклик деструктора під час завершення життя об’єкта.
- Правила оголошення деструктора.
- Деструктори та робота з динамічною пам’яттю.
- Особливості деструкторів у разі використання **композиції**

3. Вкладені класи:

- Призначення та приклади використання вкладених класів.
- Область видимості вкладеного класу.
- Доступ вкладеного класу до членів зовнішнього класу.
- Сфери застосування: інкапсуляція службових структур, допоміжні класи, реалізація ітераторів.

4. Композиція та агрегація класів:

- Композиція – включення об’єкта одного класу як члена іншого.
- Агрегація – зв’язок між класами через вказівники/посилання.
- Приклади практичного використання композиції та агрегації :
- Особливості реалізації відношень між класами у C++: створення об’єктів-складових у конструкторі, правильне використання деструкторів.
- Взаємодія з принципами інкапсуляції та повторного використання коду.

 ***Запитання для самоконтролю за темою 2***

1. Що таке конструктор у класі C++ і для чого він використовується?
2. Які існують види конструкторів у C++? Наведіть приклади.
3. У чому полягає відмінність між конструктором за замовчуванням і конструктором з параметрами?
4. Що таке конструктор копіювання? Чим відрізняється поверхневе копіювання від глибокого?
5. Для чого використовують список ініціалізації у конструкторі?
6. Що таке деструктор, коли він викликається і яку роль відіграє?

7. У чому різниця між оператором присвоєння (operator=) і конструктором копіювання?
8. Що таке вкладений клас? Як визначається його область видимості?
9. Чим відрізняється композиція від агрегації? Наведіть приклади.
10. Як впливають конструктори та деструктори на життєвий цикл об’єктів у випадку композиції?

Тема 3. Статичний поліморфізм. Перевантаження функцій. Перевантаження операторів

1. Статичний поліморфізм:

- Поняття поліморфізму. Відмінність статичного поліморфізму (вирішення під час компіляції) від динамічного (вирішення під час виконання).
- Основні засоби реалізації статичного поліморфізму у C++:

2. Перевантаження функцій:

- Поняття та призначення.
- Умови правильного перевантаження: різна кількість параметрів; різні типи параметрів; неможливість перевантаження лише за типом, що повертається.
- Приклади перевантаження функцій у бібліотеках стандартного C++.
- Проблеми неоднозначності при перевантаженні та способи їх уникнення.

3. Перевантаження конструкторів:

- Можливість створення кількох конструкторів з різними параметрами.
- Приклади використання (ініціалізація об’єктів у різних сценаріях).
- Виклик одного конструктора з іншого (C++11 – *delegating constructors*).

4. Перевантаження операторів:

- Загальні принципи: синтаксис, ключове слово operator.
- Можливості та обмеження на перевантаження операторів
- Рекомендації щодо «доречності» перевантаження (наприклад, математика, робота з контейнерами).

- Унарні та бінарні оператори
- Вибір між перевантаженням як методу класу чи зовнішньої функції (friend)
- Перевантаження операторів ==, !=, <, <=, >, >=.
- Приклади для власних структур (наприклад, клас «Дата» або «Комплексне число»).
- Оператор присвоєння
- Префіксна і постфіксна форми операторів інкрементації і декрементації. Особливості синтаксису перевантаження (додатковий параметр int для постфіксної форми). Приклади використання (ітератори, лічильники)

Запитання для самоконтролю за темою 3

1. Що таке статичний поліморфізм і чим він відрізняється від динамічного поліморфізму?
2. У яких випадках доцільно використовувати перевантаження функцій?
3. Які правила потрібно враховувати при перевантаженні функцій у C++?
4. Що таке неоднозначність при перевантаженні функцій і як її можна уникнути?
5. Чим відрізняється перевантаження конструкторів від звичайного перевантаження функцій?
6. Які оператори в C++ можна перевантажити, а які – ні? Наведіть приклади.
7. Як реалізуються унарні та бінарні оператори у класах?
8. У чому полягає різниця між префіксною і постфіксною формами інкрементації/декрементації при перевантаженні?
9. Що таке правило “Великої трійки” і яке відношення воно має до оператора присвоєння?
10. Які потенційні ризики чи недоліки може мати неправильне використання перевантажених операторів у програмах?

Тема 4. Успадкування. Конструктори та деструктори при успадкуванні

1. Основи успадкування:

- Поняття успадкування як механізму повторного використання коду.
- Принцип успадкування: створення похідного класу на основі існуючого базового.
- Похідні класи: як додають нові атрибути і методи або змінюють поведінку базового класу.
- Роль успадкування у створенні ієрархій класів та поліморфізму.

2. Специфікатори доступу при успадкуванні:

- Види специфікаторів при оголошенні успадкування: public (зберігає видимість public/protected членів); protected (public члени базового класу стають protected); private(усі члени базового класу стають private у похідному).
- Порівняння зі специфікаторами доступу всередині класу (public, private, protected).
- Захищені члени класу (protected): доступні у похідних класах, але приховані від зовнішнього коду.
- Управління доступом до базового класу: коли потрібне обмеження успадкування.

3. Конструктори і деструктори при успадкуванні:

- Конструктори базового класу не успадковуються автоматично.
- Порядок виклику конструкторів
- Передача параметрів конструктору базового класу через список ініціалізації.
- Виклик деструкторів:
- Приклад життєвого циклу об’єкта у спадковій ієрархії.

4. Ієрархічне успадкування:

- Побудова дерева класів (наприклад, клас Транспорт → Автомобіль → Електромобіль).
- Виклик конструкторів в ієрархії: послідовне виконання конструкторів від базового до найнижчого рівня.
- Використання ключового слова virtual для деструкторів – запобігання витокам пам’яті при роботі з поліморфізмом.

5. Множинне успадкування:

- Поняття та приклади (наприклад, клас Студент-Спортсмен, який успадковується від класів Студент і Спортсмен).
- Порядок виклику конструкторів при множинному успадкуванні.
- Передача параметрів до конструкторів базових класів.
- Проблема «ромбовидного успадкування» (diamond problem).
- Використання віртуального успадкування (virtual base class) для уникнення дублювання базового класу.

6. Розширені можливості успадкування:

- Використання ключових слів override, final (C++11+) для уточнення перевизначення методів.
- Приховування методів базового класу у похідному (using Base::method для відновлення доступу).
- Заборона успадкування (final class).
- Практичні рекомендації: коли доцільно застосовувати успадкування, а коли краще використовувати композицію.

 ***Запитання для самоконтролю за темою 4***

1. З якою метою успадкування використовується в об’єктно-орієнтованому програмуванні?
2. Які є види успадкування у C++ і чим вони відрізняються (public, protected, private)?

3. Яка різниця між protected-членами та private-членами класу?
4. У якому порядку викликаються конструктори при створенні об’єкта похідного класу?
5. У якому порядку викликаються деструктори при знищенні об’єкта похідного класу?
6. Як передати параметри конструктору базового класу з конструктора похідного класу?
7. Чому важливо оголошувати віртуальний деструктор у базовому класі, якщо він використовується поліморфно?
8. Що таке ієрархічне успадкування? Наведіть приклад.
9. Які проблеми можуть виникнути при множинному успадкуванні? Як їх вирішують?
10. У яких випадках краще замість успадкування використовувати композицію?

Тема 5. Віртуальні функції. Вказівники на похідні класи. Абстрактні класи

1. Динамічний поліморфізм:

- Поняття динамічного поліморфізму (вирішення виклику функції під час виконання, а не на етапі компіляції).
- Відмінність від статичного поліморфізму.
- Переваги: розширюваність програм, можливість роботи з колекціями об’єктів різних типів через один інтерфейс.

2. Віртуальні функції:

- Призначення віртуальних функцій.
- Синтаксис оголошення (virtual перед методом базового класу).
- Виклик функцій через вказівники та посилання на базовий клас.
- Поняття пізнього зв’язування і таблиці віртуальних функцій (vtable).
- Приклади: реалізація однакового інтерфейсу для різних похідних класів.

3. Вказівники на похідні класи:

- Використання базового вказівника для збереження адреси об’єкта похідного класу.
- Віртуальні функції та правильний виклик методів у такій ситуації.
- Приклади: масив або контейнер базових вказівників на об’єкти різних похідних класів.
- Небезпека роботи без віртуального деструктора (витік пам’яті).

4. Абстрактні класи:

- Визначення абстрактного класу як класу, що містить хоча б одну чисто віртуальну функцію.
- Неможливість створення об’єктів абстрактного класу.
- Використання абстрактних класів для побудови інтерфейсів у C++.

5. Віртуальні базові класи:

- Проблема ромбовидного успадкування
- Використання ключового слова virtual під час оголошення базового класу
- Запобігання дублюванню членів базового класу у множинному успадкуванні.

 ***Запитання для самоконтролю за темою 5***

1. У чому полягає відмінність між статичним і динамічним поліморфізмом у C++?
2. Як правильно оголосити віртуальну функцію в базовому класі?
3. Що таке пізнє зв’язування і чим воно відрізняється від раннього?
4. Яку роль відіграє таблиця віртуальних функцій (vtable) у механізмі поліморфізму?
5. Чому важливо робити деструктори віртуальними у базових класах, що мають похідні?
6. Що таке абстрактний клас? Чи можна створювати його об’єкти?

7. Як оголошується чисто віртуальна функція і яка її роль?
8. Що станеться з похідним класом, якщо він не реалізує чисто віртуальну функцію базового класу?
9. Наведіть приклад використання вказівника на базовий клас для роботи з об’єктами похідних класів.
10. Що таке віртуальний базовий клас і як він допомагає вирішити проблему ромбовидного успадкування?

Тема 6. Оброблення виняткових ситуацій

1. Вступ до оброблення помилок:

- Поняття **виняткової ситуації (exception)** у програмуванні.
- Розпізнавання ситуацій, які потребують обробки помилок.
- Класифікація помилок: помилки введення/виведення; логічні помилки; помилки динамічної пам’яті; критичні помилки системи.
- Важливість управління ресурсами (RAII – Resource Acquisition Is Initialization).

2. Генерація та перехоплення виняткових ситуацій:

- Використання оператора throw для створення винятку.
- Передача виняткових об’єктів (класів чи примітивних типів).
- Вибір правильного типу для винятку (клас чи стандартний тип).
- Використання блоку try для захисту коду.
- Оператор catch для перехоплення винятків.
- Можливість декількох операторів catch для різних типів винятків.
- Оброблення похідних класів виняткових ситуацій (успадкування).
- Використання ключового слова throw; у середині блоку catch. Сценарії повторного кидання для передавання винятку на вищий рівень обробки

3. Функції terminate() і unexpected():

- Функція terminate() викликається, коли неможливо продовжити виконання програми після винятку.

- Функція unexpected() викликається, якщо функція кинула виняток, який не був заявлений у її специфікації (старий механізм exception specification).
- Можливість встановлення власних обробників через set_terminate() та set_unexpected().

4. Класи стандартної бібліотеки:

- std::exception – базовий клас для всіх стандартних винятків.
- std::bad_exception – обробка непередбачених винятків.
- Використання методів what() для отримання опису помилки.

5. Виняткові ситуації та успадкування:

- Створення власних класів винятків через успадкування від std::exception.
- Поліморфне перехоплення похідних класів виняткових ситуацій.
- Рекомендації: обробляти спершу конкретні винятки, потім – базові.

Запитання для самоконтролю за темою 6

1. Що таке виняткова ситуація (exception) у C++ і коли її доцільно використовувати?
2. Які типи помилок можна виділити під час класифікації виняткових ситуацій?
3. Як у C++ генерується виняткова ситуація?
4. Яку роль відіграють блоки try і catch у обробленні винятків?
5. Як можна перехоплювати кілька типів винятків у одному блоці try?
6. Що таке повторне генерування виняткової ситуації і як воно реалізується?
7. Які функції відповідають за критичне завершення програми при винятках (terminate() і unexpected()) і як вони працюють?
8. Які особливості класів стандартної бібліотеки std::exception і std::bad_exception?
9. Як правильно реалізовувати власні класи винятків через успадкування?

10. У якому порядку слід перехоплювати похідні та базові виняткові класи для уникнення проблем поліморфізму?

Тема 7. Шаблони і успадкування. Бібліотека стандартних шаблонів STL.

1. Основи шаблонів:

- Поняття шаблону (template) у C++: узагальнення функцій та класів для роботи з різними типами даних.
- Простий шаблон: синтаксис для функції та класу.
- Параметризовані функції та класи: передача типу як параметра.
- Правила ототожнення при перевантаженні параметризованих функцій (template argument deduction).
- Приклади використання: функції обміну значень, обчислення максимуму/мінімуму.

2. Шаблони і успадкування:

- Можливість створення похідних класів від шаблонних класів.
- Використання шаблонів із дружніми функціями (friend).
- Узагальнені класи з двома або більше узагальненими типами.
- Застосування аргументів у шаблонних класах для керування поведінкою об’єкта.

3. Бібліотека стандартних шаблонів (STL):

- Склад STL: контейнери, ітератори, алгоритми, функції-об’єкти (functors).
- Предикати та функції-об’єкти: об’єкти, які можна викликати як функцію.
- Алокатори: керування динамічною пам’яттю у контейнерах STL.

4. Контейнери STL:

- Стек (stack) – принцип LIFO, основні методи: push, pop, top.
- Черга (queue) – принцип FIFO, методи: push, pop, front, back.
- Пріоритетна черга (priority_queue) – елементи витягуються за пріоритетом.

- Контейнери-послідовності: vector – динамічний масив; deque – двостороння черга;
- Асоціативні контейнери: set – унікальні елементи в порядку сортування; map – відображення ключ→значення.

5. Ітератори:

- Поняття ітератора: абстрактний покажчик на елемент контейнера.
- Типи ітераторів:
- вхідні (input), вихідні (output), прямі (forward), двонаправлені (bidirectional), випадковий доступ (random access).
- Методи роботи з ітераторами: begin(), end(), ++, --, розіменування *it.
- Застосування ітераторів для обходу контейнерів та використання з алгоритмами STL.

6. Узагальнені алгоритми STL:

- Використання функцій та об’єктів-функцій із контейнерами.
- Приклади алгоритмів: сортування (sort), пошук (find), копіювання (copy).
- Застосування алгоритмів у комбінації з ітераторами для різних контейнерів.

 ***Запитання для самоконтролю за темою 7***

1. Що таке шаблон (template) у C++ і для чого він використовується?
2. Як створити простий шаблон функції та шаблон класу? Наведіть приклади.
3. Що таке параметризовані функції і як працює deduction типів у шаблонах?
4. Як застосовувати шаблони та успадкування одночасно?
5. Що таке явна спеціалізація класу і коли вона потрібна?
6. Які основні компоненти бібліотеки STL і яку роль вони виконують?
7. Що таке контейнер у STL? Наведіть приклади контейнерів-послідовностей та асоціативних контейнерів.
8. Що таке ітератор і які типи ітераторів існують у STL?

9. Які алгоритми STL ви знаєте і як їх можна використовувати з ітераторами?
10. Як використовуються предикати та функції-об’єкти у поєднанні з контейнерами STL?

II СЕМЕСТР

Тема 8. Основи C# і платформа .NET

1. Зв’язок C# із .NET Framework:

- Об’єктно-орієнтована мова програмування C#.
- Платформи для розробки кросплатформених додатків .NET Framework / .NET Core / .NET 5+ –.
- Загальна інфраструктура виконання CLI.
- Система загальних типів CTS, що забезпечує узгодженість типів між мовами.
- CLS – набір правил для сумісності коду між мовами .NET.
- Простори імен (namespaces).

2. Основи C#: змінні, типи, оператори:

- Змінні та типи даних: value types (int, double, bool, struct) та reference types (class, string, array).
- Літерали для чисел, символів, рядків та логічних значень.
- Перетворення типів: явне та неявне (casting, Convert, Parse, TryParse).
- Умовні оператори: if, else, switch.
- Циклічні оператори: for, while, do while, foreach.
- Оператори переходу: break, continue, return, goto.

3. Масиви:

- Одновимірні та багатовимірні масиви ([,], [][]).
- Клас Array: методи Length, Rank, GetLength, Sort, Reverse.
- Масиви як параметри функцій, повернення масивів.
- Застосування масивів для зберігання колекцій однотипних даних.

5. Рядкові змінні String та StringBuilder:

- Тип string: immutable (незмінний), методи Substring, IndexOf, Replace, Split, Trim.
- StringBuilder: mutable рядки, ефективні при частих змінах рядків (Append, Insert, Remove).
- Форматуючі рядки: інтерполяція ($""), String.Format, PadLeft, PadRight.

6. Регулярні вирази:

- Використання класу Regex для пошуку, перевірки та заміни рядків.
- Основні методи: IsMatch, Match, Matches, Replace.
- Приклади застосування: перевірка формату електронної пошти, номерів телефонів, виділення певних патернів у тексті.

 ***Запитання для самоконтролю за темою 8***

1. Що таке .NET Framework і як C# пов’язаний із цією платформою?
2. Яка роль CLI, CTS і CLS у платформі .NET?
3. Чим відрізняються value types від reference types у C#?
4. Які є способи перетворення типів у C#? Наведіть приклади.
5. Які умовні оператори та цикли підтримує C#? Наведіть приклади їх використання.
6. Як оголошуються та ініціалізуються масиви в C#?
7. У чому відмінність між одновимірними, багатовимірними та зубчастими масивами?
8. Що таке структури (struct) і чим вони відрізняються від класів?
9. Як працюють рядки (string) та StringBuilder? Коли доцільно використовувати StringBuilder?
10. Як застосовуються регулярні вирази (Regex) у C#? Наведіть приклади методів класу Regex.

Тема 9. Реалізація принципів ООП мовою C#. Класи і об'єкти. Перевантаження

1. Класи і об’єкти:

- Клас як шаблон для створення об’єктів, що описує стан (поля) та поведінку (методи).
- Клас Object – базовий клас для всіх типів у C#. Методи: ToString(), Equals(), GetHashCode(), GetType().
- Створення об’єктів: через конструктори, ключове слово new.
- Методи класу – реалізація поведінки об’єкта.
- Конструктори та деструктори:
- Ключове слово this – посилання на поточний об’єкт.
- Доступ до членів класу: через модифікатори доступу public, private, protected, internal.

2. Модифікатори та параметри:

- Модифікатори параметрів: ref, out, in.
- Необов’язкові та іменовані аргументи при виклику методів.
- Рекурсія – методи, які викликають самі себе.
- Ключове слово static – члени класу, що належать самому класу, а не об’єктам.

3. Індексатори і властивості:

- Індексатори – доступ до елементів об’єкта як до масиву.
- Властивості (properties) – контрольований доступ до полів класу через аксесори get і set.
- Модифікатори доступу в аксесорах – можливість обмежити доступ до get або set.
- Використання властивостей та індексаторів для інкапсуляції даних
- Статичні методи та поля для загальних ресурсів і допоміжних функцій

4. Перевантаження методів і операторів:

- Перевантаження методів класу.
- Перевантаження індексаторів (різні сигнатури для доступу до даних).
- Основи перевантаження операторів.

Запитання для самоконтролю за темою

1. Що таке клас і об’єкт у C# і як вони пов’язані?
2. Яку роль відіграє клас Object у C#? Назвіть основні його методи.
3. Як створюються об’єкти класу і що таке конструктор та деструктор?
4. Для чого використовується ключове слово this у методах класу?
5. Які є модифікатори доступу для членів класу і як вони впливають на видимість?
6. Що таке індексатори і як вони працюють у класах C#?
7. Як визначаються властивості (properties) і які існують аксесори get і set?
8. Що таке перевантаження методів і які правила його застосування?
9. Як перевантажуються оператори у C# і які оператори можна перевантажувати?
10. Що таке оператори перетворення implicit і explicit, і коли їх застосовують?

Тема 10. Успадкування та поліморфізм у C#. Інтерфейси

1. Основи успадкування:

- Механізм успадкування у C#.
- Захищений доступ (protected), що дозволяє похідним класам доступ до членів базового класу.
- Виключення успадкування (sealed) – заборона подальшого успадкування класу.
- Конструктори та успадкування: виклик базового конструктора через : base(...).
- Приховування імен (new) – заміна членів базового класу у похідному.

2. Поліморфізм:

- Віртуальні методи, властивості і індексатори (virtual).
- Посилання на базовий клас для роботи з об’єктами похідних класів.
- Абстрактні класи: неможливість створення об’єктів, наявність абстрактних методів.

3. Інтерфейси:

- Інтерфейс (interface) стуктура, що визначає методи, властивості, індексатори без реалізації.
- Інтерфейсні посилання – доступ до об’єкта через тип інтерфейсу.
- Інтерфейсні властивості та індексатори – доступ до даних через контракт.
- Успадкування інтерфейсів. Явна реалізація інтерфейсу.
- Реалізація декількох інтерфейсів у класах та структурах.
- Використання явної реалізації інтерфейсів для уникнення конфліктів імен.

 ***Запитання для самоконтролю за темою 10***

1. Що таке успадкування у C# і яка його роль у повторному використанні коду?
2. Яку роль відіграє модифікатор доступу protected у механізмі успадкування?
3. Що означає ключове слово sealed і коли його застосовують?
4. Як викликається конструктор базового класу з похідного класу?
5. Що таке віртуальні методи, властивості та індексатори, і як їх перевизначати у похідних класах?
6. Що таке абстрактний клас і чим він відрізняється від звичайного класу?
7. Що таке інтерфейс (interface) і як його використовують у C#?
8. Як реалізується явна реалізація інтерфейсу і для чого вона потрібна?
9. Чим відрізняються структури (struct) від класів і які обмеження у них щодо успадкування?

10. Як посилання на базовий клас дозволяє реалізувати поліморфізм при роботі з об’єктами похідних класів?

Тема 11. Оброблення винятків засобами C#. Делегати та лямбда-вирази. Узагальнення

1. Оброблення винятків у C#:

- Основи оброблення винятків: використання блоків try, catch та finally.
- Клас Exception – базовий клас для всіх винятків у C#. Методи: Message, StackTrace, InnerException.
- Винятки, пов’язані з пошкодженим станом (Corrupted State Exceptions, CSE).
- Ключові слова checked і unchecked – контроль перевірки арифметичних операцій на переповнення.

2. Делегати та лямбда-вирази:

- Делегати. Груповий виклик делегатів.
- Коваріантність і контраваріантність делегатів.
- Action і Func – стандартні узагальнені делегати для методів без і з поверненим значенням.
- Анонімні методи – методи без імені, які можна присвоїти делегату.
- Лямбда-вирази – компактна форма анонімного методу: (x, y) => x + y.

3. Узагальнення (Generics):

- Узагальнені класи, методи та структури – параметризовані типи для підвищення гнучкості.
- Обмежені типи: where T : class, where T : struct, where T : new(), обмеження інтерфейсами.
- Ієрархії узагальнених класів.
- Узагальнені делегати та інтерфейси.

- Коваріантність і контраваріантність в узагальненнях – керування напрямком типів для інтерфейсів і делегатів.
- Модифікація узагальнених методів – адаптація під різні типи параметрів.

4. Потоки вводу-виводу (I/O):

- Стандартні класи потоків: Stream, FileStream, StreamReader, StreamWriter.
- Уведення та виведення з потоками – читання і запис даних у файли та інші потоки.
- Маніпулятори потоків – форматування даних при введенні та виведенні.

Запитання для самоконтролю за темою 11

1. Як реалізується оброблення винятків у C# за допомогою блоків try, catch та finally?
2. Яку роль відіграє клас Exception і які основні властивості він надає?
3. Що таке Corrupted State Exceptions (CSE) і коли вони виникають?
4. Для чого використовуються ключові слова checked та unchecked?
5. Що таке делегат і як він використовується для виклику методів?
6. Як працює груповий виклик делегатів і що відбувається при наявності декількох методів у делегаті?
7. Що таке анонімні методи і лямбда-вирази? Наведіть приклади їх використання.
8. Що таке узагальнені класи та методи, і як обмежуються типи за допомогою where?
9. Що таке коваріантність і контраваріантність у делегатах та узагальненнях?
10. Які стандартні класи потоків існують у C# і як вони використовуються для вводу/виводу даних?

ЛАБОРАТОРНІ ЗАНЯТТЯ

1 СЕМЕСТР

Лабораторна робота 1. Створення та підключення класу в середовищі програмування (MS Visual Studio, VS Code). Оголошення та будова класу.

Мета роботи: Ознайомитися зі створенням та підключенням класів у середовищах програмування MS Visual Studio та VS Code, навчитися оголошувати класи, їхні поля і методи, організовувати структуру проекту для розробки об’єктно-орієнтованих програм.

 ***Очікувані знання та навички:***

- Знати основні поняття об’єктно-орієнтованого програмування: клас, об’єкт, інкапсуляція, синтаксис оголошення класів, полів та методів у C++ або C#.
- Знати принципи організації проекту та підключення класів у середовищі програмування.
- Знати роль модифікаторів доступу (public, private, protected).
- Вміти створити клас в проекті та підключення його до інших частин програми.
- Вміти реалізовувати ініціалізацію об’єктів і виклик методів класу.
- Використовувати поля і методи для моделювання поведінки об’єкта.
- Розробляти прості програми з організованою структурою класів.

Лабораторна робота 2. Робота з полями та методами. Об’єкти класу як параметри методів.

Мета роботи: ознайомитися з роботою полів та методів класу, навчитися передавати об’єкти як параметри методів, використовувати різні типи методів,

дружні функції, а також реалізовувати доступ до полів через сетери та гетери для організації безпечної роботи з даними об’єктів.

 ***Очікувані знання та навички:***

- Знати призначення та особливості різних типів методів класу (звичайні, статичні, дружні).
- Знати принципи використання сетерів і геттерів для контролю доступу до полів класу.
- Знати способи передачі об’єктів класу як параметрів методам.
- Вміти оголошувати та реалізовувати методи класу різних типів, застосовувати сетери та гетери для безпечного доступу до даних.
- Вміти створювати дружні функції для взаємодії з полями класу.
- Розробляти прості програми з коректною передачею об’єктів у методи та організацією доступу до їхніх полів.

Лабораторна робота 3. Конструктори та деструктори.

Мета роботи: ознайомитися з роботою конструкторів та деструкторів у класах, навчитися ініціалізувати об’єкти різними способами та організовувати звільнення ресурсів після завершення роботи об’єкта для забезпечення коректної роботи об’єктно-орієнтованої програми.

 ***Очікувані знання та навички:***

- Знати призначення конструкторів і деструкторів у класах.
- Знати види конструкторів (за замовчуванням, копіювання, ініціалізації) та правила виклику конструкторів і деструкторів у програмі.
- Вміти оголошувати та реалізовувати різні конструктори класу.
- Вміти створювати об’єкти класу за допомогою різних конструкторів.
- Вміти контролювати ініціалізацію та звільнення ресурсів у деструкторах.
- Розробляти прості програми з правильною ініціалізацією об’єктів та коректним звільненням ресурсів.

Лабораторна робота 4. Реалізація перевантаження операцій для класів.

Мета роботи: ознайомитися з механізмом перевантаження операторів у класах, навчитися реалізовувати перевантаження арифметичних, логічних та операторів відношення, а також застосовувати їх для підвищення зручності роботи з об’єктами та розробки більш інтуїтивно зрозумілих програм.

 ***Очікувані знання та навички:***

- Знати принципи перевантаження операторів у класах.
- Вміти реалізовувати перевантаження унарних та бінарних операторів, операторів відношення (==, !=, <, >, <=, >=).
- Вміти створювати префіксні та постфіксні форми операторів інкрементації та декрементації.
- Вміти перевантажувати оператори присвоєння та логічні оператори.
- Розробляти програми, де об’єкти класів можна використовувати у виразах як звичайні типи даних завдяки перевантаженим операціям.

Лабораторна робота 5. Робота з вкладеними класами.

Мета роботи: ознайомитися з особливостями створення та використання вкладених класів у програмі, навчитися організовувати ієрархію класів, визначати області видимості та взаємодіяти з внутрішніми об’єктами для підвищення структурованості та інкапсуляції даних у програмі.

 ***Очікувані знання та навички:***

- Знати правила оголошення вкладених класів та їхні області видимості, особливості доступу до членів внутрішнього класу з базового та зовнішнього класу.
- Вміти створювати вкладені класи та об’єкти цих класів, організовувати взаємодію між зовнішнім і внутрішнім класами.

- Вміти застосовувати вкладені класи для структурування даних і реалізації складних об’єктів.
- Розробляти програми з правильною організацією ієрархії класів і забезпеченням інкапсуляції.

Лабораторна робота 6. Композиція та агрегація класів на C++.

Мета роботи: Ознайомитися з принципами композиції та агрегації класів у C++, навчитися створювати складні об’єкти за допомогою включення інших класів, а також організовувати взаємодію між об’єктами для підвищення структурованості та повторного використання коду.

 ***Очікувані знання та навички:***

- Знати різницю між композицією та агрегацією класів.
- Знати правила оголошення класів та включення об’єктів інших класів.
- Вміти створювати об’єкти класів у складі інших класів, організовувати взаємодію між складовими об’єктами.
- Вміти застосовувати композицію та агрегацію для побудови більш складних об’єктів.
- Розробляти програми з правильною організацією зв’язків між класами та повторним використанням коду.

Лабораторна робота 7. Реалізація відкритого успадкування засобами C++.

Мета роботи: ознайомитися з механізмом відкритого успадкування у C++, навчитися створювати похідні класи, реалізовувати доступ до членів базового класу та застосовувати поліморфізм для побудови гнучкої та повторно використовуваної структури об’єктів.

 ***Очікувані знання та навички:***

- Знати принципи відкритого успадкування та специфікатори доступу (public, protected, private).

- Знати правила виклику конструкторів і деструкторів базового та похідного класів.
- Знати особливості реалізації простого успадкування.
- Вміти створювати похідні класи та оголошувати їх члени.
- Вміти викликати методи базового класу з похідного класу, застосовувати поліморфізм для роботи з об’єктами базового та похідного класів.
- Розробляти програми з правильною організацією ієрархії класів та використанням успадкування для повторного використання коду.

Лабораторна робота 8. Просте та множинне успадкування.

Мета роботи: ознайомитися з механізмом простого та множинного успадкування у C++, навчитися створювати похідні класи з одного або кількох базових класів, організовувати виклик конструкторів і деструкторів та застосовувати успадкування для побудови гнучкої ієрархії об’єктів.

 ***Очікувані знання та навички:***

- Знати принципи простого та множинного успадкування.
- Знати особливості виклику конструкторів і деструкторів у ієрархії класів, способи управління доступом до членів базових класів у похідних класах.
- Вміти створювати похідні класи на основі одного або кількох базових класів.
- Вміти застосовувати успадкування для повторного використання коду, організовувати ієрархію класів для побудови складних об’єктів.
- Розробляти програми з правильною структурою класів та коректним використанням простого і множинного успадкування.

Лабораторна робота 9. Віртуальні функції. Робота з абстрактними класами засобами C++.

Мета роботи: Ознайомитися з механізмом віртуальних функцій та абстрактних класів у C++, навчитися реалізовувати динамічний поліморфізм, перевизначати методи базового класу у похідних класах та практично використовувати абстрактні класи.

 ***Очікувані знання та навички:***

- Знати принципи роботи віртуальних функцій та механізм пізнього зв’язування.
- Знати призначення абстрактних класів і чисто віртуальних функцій.
- Вміти створювати похідні класи та перевизначати віртуальні методи, застосовувати абстрактні класи.
- Вміти працювати з вказівниками на базовий клас для реалізації динамічного поліморфізму.
- Розробляти програми з використанням віртуальних функцій і абстрактних класів для підвищення гнучкості та повторного використання коду.

Лабораторна робота 10. Інтерфейси.

Мета роботи: ознайомитися з використанням інтерфейсів у C++ через чисто віртуальні класи, навчитися створювати загальні інтерфейси для об’єктів, реалізовувати поліморфізм і забезпечувати єдиний контракт для різних класів у програмі.

 ***Очікувані знання та навички:***

- Знати принципи створення інтерфейсів за допомогою чисто віртуальних класів у C++.
- Знати особливості реалізації інтерфейсних методів у похідних класах.
- Вміти створювати інтерфейсні класи та похідні класи, що їх реалізують, застосовувати інтерфейсні посилання та вказівники для роботи з різними об’єктами.
- Вміти реалізовувати множинне успадкування інтерфейсів.

- Розробляти програми з використанням інтерфейсів для забезпечення гнучкої ієрархії класів та поліморфної поведінки.

Лабораторна робота 11. Використання класів при опрацюванні виняткових ситуацій.

Мета роботи: ознайомитися з механізмом оброблення виняткових ситуацій у C++, навчитися використовувати класи для генерації та перехоплення винятків, реалізовувати повторне генерування винятків та організовувати безпечну роботу програми при виникненні помилок.

 ***Очікувані знання та навички:***

- Знати принципи оброблення виняткових ситуацій у C++.
- Знати призначення класів exception, bad_exception та їхні основні властивості.
- Знати правила генерації, перехоплення та повторного генерування винятків.
- Вміти створювати власні класи виняткових ситуацій, використовувати блоки try, catch для контролю виконання програми.
- Вміти застосовувати кілька операторів catch для обробки різних типів винятків.
- Розробляти програми з правильною організацією оброблення виняткових ситуацій і безпечним звільненням ресурсів об’єктів.

Лабораторна робота 12. Контейнери як параметри.

Мета роботи: ознайомитися з використанням стандартних контейнерів STL у C++, навчитися передавати контейнери як параметри функцій і методів класів, а також ефективно працювати з їхніми елементами для організації зручної та гнучкої обробки даних.

 ***Очікувані знання та навички:***

- Знати основні типи стандартних контейнерів STL (vector, deque, list, set, map) та їх особливості.
- Знати правила передачі контейнерів як параметрів функцій і методів класів.
- Вміти працювати з ітераторами для доступу до елементів контейнера.
- Вміти застосовувати функції STL для обробки даних у контейнерах.
- Розробляти програми з правильною організацією передачі та обробки контейнерів у функціях і методах класів.

Лабораторна робота 13. Шаблони класів та шаблони функцій. Бібліотека STL

Мета роботи: Ознайомитися з використанням шаблонів класів та функцій у C++, навчитися створювати узагальнені класи та функції для роботи з різними типами даних, а також застосовувати можливості стандартної бібліотеки STL для організації ефективної обробки даних.

 ***Очікувані знання та навички:***

- Знати принципи створення шаблонів класів і функцій, правила явної та неявної спеціалізації шаблонів.
- Вміти оголошувати та реалізовувати узагальнені класи та функції.
- Вміти застосовувати шаблони для роботи з різними типами даних без дублювання коду.
- Знати основні компоненти бібліотеки STL та їх призначення (контейнери, ітератори, алгоритми, функціональні об’єкти).
- Вміти використовувати контейнери STL у поєднанні з шаблонними класами та функціями.
- Розробляти програми з ефективною організацією даних та повторним використанням узагальненого коду.

2 СЕМЕСТР

Лабораторна робота 1. Основи програмування на C#. Розгалуження. Цикли

Мета роботи: Ознайомитися з основами програмування на C#, навчитися використовувати умовні оператори та цикли для організації керованого виконання програми, а також розробляти алгоритми з повторюваними та умовними обчисленнями.

 ***Очікувані знання та навички:***

- Знати типи даних, змінні та літерали в C#.
- Знати принципи використання умовних та циклічних операторів
- Вміти будувати прості алгоритми з розгалуженнями та повтореннями.
- Вміти застосовувати оператори переходу (break, continue, return) у циклах та умовних конструкціях.
- Вміти читати та обробляти дані користувача через змінні.
- Розробляти прості програми з правильною логікою розгалужень та циклів.

Лабораторна робота 2. Основи програмування на C#. Масиви. Рядки. Структури.

Мета роботи: Ознайомитися з роботою з масивами, рядками та структурами в C#, навчитися створювати та обробляти одномірні та багатовимірні масиви, використовувати методи роботи з рядками, а також визначати та застосовувати структури для зберігання даних різних типів.

 ***Очікувані знання та навички:***

- Знати правила оголошення та ініціалізації масивів (одномірних та багатовимірних), методи роботи з масивами та клас Array.
- Вміти виконувати обхід, вставку, видалення та пошук елементів масивів.

- Знати особливості роботи з рядками (String, StringBuilder) та форматування рядків.
- Вміти створювати та використовувати структури для групування даних, передавати масиви та структури як параметри методів.
- Розробляти програми з правильною організацією даних у масивах, рядках та структурах.

Лабораторна робота 3. Робота з класами та об’єктами засобами C#.

Мета роботи: Ознайомитися з основами об’єктно-орієнтованого програмування в C#, навчитися створювати класи та об’єкти, використовувати поля, методи та властивості для моделювання поведінки об’єктів, а також організовувати взаємодію об’єктів у програмі.

 ***Очікувані знання та навички:***

- Знати синтаксис оголошення класів, полів, методів і властивостей у C#.
- Знати призначення ключового слова this та модифікаторів доступу (public, private, protected).
- Вміти створювати об’єкти класів і викликати їхні методи та властивості; передавати об’єкти як параметри методів та отримувати об’єкти як результати.
- Вміти використовувати індексатори та властивості для доступу до даних об’єкта.
- Розробляти прості програми з правильною організацією класів і об’єктів для моделювання поведінки системи.

Лабораторна робота 4. C#. Конструктори та операції.

Мета роботи: ознайомитися з використанням конструкторів у C# для ініціалізації об’єктів, навчитися реалізовувати перевантаження методів та

операторів для класів, а також застосовувати різні операції над об’єктами для підвищення гнучкості та зручності програмування.

 ***Очікувані знання та навички:***

- Знати принципи оголошення та використання конструкторів (без параметрів, з параметрами, копіювання).
- Вміти створювати об’єкти та ініціалізувати їх через конструктори.
- Вміти перевантажувати методи класу для реалізації різних варіантів виконання.
- Розуміти використання конструкторів для контролю початкового стану об’єкта.
- Розробляти програми з правильною ініціалізацією об’єктів та застосуванням перевантажених операцій.

Лабораторна робота 5. Реалізація композиції класів засобами C#. Вкладені класи.

Мета роботи: ознайомитися з реалізацією композиції класів та використанням вкладених класів у C#, навчитися створювати складні об’єкти через включення інших класів і організовувати взаємодію між об’єктами для підвищення структурованості та інкапсуляції даних у програмі.

 ***Очікувані знання та навички:***

- Знати принципи реалізації композиції класів у C#; правила оголошення вкладених класів та їхніх областей видимості.
- Вміти створювати об’єкти класів у складі інших класів.
- Вміти організовувати взаємодію між зовнішнім і вкладеним класами.
- Вміти застосовувати властивості, методи та поля для доступу до об’єктів вкладених класів.
- Розробляти програми з правильною організацією структури класів та об’єктів для моделювання складних систем.

Лабораторна робота 6. Робота з абстрактними класами та інтерфейсами засобами C#.

Мета роботи: ознайомитися з використанням абстрактних класів та інтерфейсів у C#, навчитися створювати узагальнені інтерфейси для об'єктів, реалізовувати поліморфізм і забезпечувати єдиний контракт для різних класів у програмі.

 ***Очікувані знання та навички:***

- Знати принципи оголошення абстрактних класів та чисто віртуальних методів у C#.
- Знати правила оголошення інтерфейсів та їх реалізації у класах.
- Вміти створювати абстрактні класи та похідні класи, що реалізують абстрактні методи.
- Вміти реалізовувати інтерфейси та працювати з інтерфейсними посиланнями.
- Вміти застосовувати поліморфізм для об'єктів, які реалізують спільний інтерфейс або наслідують абстрактний клас.
- Розробляти програми з правильною організацією класів та інтерфейсів для гнучкої та повторно використовуваної архітектури.

Лабораторна робота 7. Опрацювання виняткових ситуацій. Серіалізація.

Мета роботи: ознайомитися з механізмом оброблення виняткових ситуацій у C# та навчитися застосовувати серіалізацію для збереження і відновлення стану об'єктів, забезпечуючи безпечну та надійну роботу програм.

 ***Очікувані знання та навички:***

- Знати принципи генерації та перехоплення виняткових ситуацій у C#.
- Знати класи винятків і механізми обробки (try, catch, finally).
- Вміти створювати власні класи винятків і повторно генерувати винятки.
- Знати основи серіалізації та десеріалізації об'єктів у C#.

- Вміти серіалізувати та десеріалізувати об’єкти для збереження стану програми.
- Вміти застосовувати оброблення винятків разом із серіалізацією для безпечного виконання програм.
- Розробляти програми з правильною організацією оброблення виняткових ситуацій і управління станом об’єктів.

Лабораторна робота 8. Узагальнені типи та колекції C#.

Мета роботи: ознайомитися з узагальненими типами (generics) та стандартними колекціями C#, навчитися створювати узагальнені класи і методи, а також ефективно працювати з колекціями для організації гнучкої та безпечної обробки даних.

 ***Очікувані знання та навички:***

- Знати принципи створення узагальнених класів, методів та інтерфейсів у C#.
- Вміти застосовувати обмеження типів (constraints) у узагальненнях.
- Знати основні колекції C# та їхні особливості.
- Вміти створювати, додавати, видаляти та шукати елементи у колекціях, використовувати ітератори та цикли foreach для обробки колекцій.
- Вміти комбінувати узагальнені типи та колекції для створення гнучких і типобезпечних структур даних.
- Розробляти програми з правильною організацією даних та узагальненими методами для підвищення повторного використання коду.

Лабораторна робота 9. Використання делегатів. Робота з потоками даних.

Мета роботи: Ознайомитися з використанням делегатів у C# для організації групових викликів методів та забезпечення гнучкої взаємодії

об’єктів, а також навчитися працювати з потоками даних для вводу та виводу інформації.

 ***Очікувані знання та навички:***

- Знати принципи оголошення та використання делегатів у C#.
- Вміти створювати делегати для групового виклику методів.
- Знати поняття коваріантності та контраваріантності делегатів.
- Вміти використовувати типи делегатів Action і Func; створювати анонімні методи та лямбда-вирази для делегатів.
- Знати основи роботи з потоками вводу та виводу (StreamReader, StreamWriter, FileStream).
- Вміти виконувати читання та запис даних у файли, обробляти дані в потоках.
- Розробляти програми з правильною організацією делегатів та потоків для ефективної обробки інформації.

ЗРАЗОК ОФОРМЛЕННЯ ЗВІТУ ПРО ВИКОНАНУ ЛАБОРАТОРНУ РОБОТУ

ЗВІТ
про виконання лабораторної роботи № 6
«Реалізація композиції класів засобами C#.»
з дисципліни
«Об’єктно орієнтоване програмування»
Студента групи КН-2025б
Петренка Петра Івановича

Мета роботи: Ознайомитися з принципами композиції та агрегації класів на C#, навчитися організовувати взаємодію між класами для підвищення структурованості та повторного використання коду.

Завдання лабораторної роботи (варіант 9)

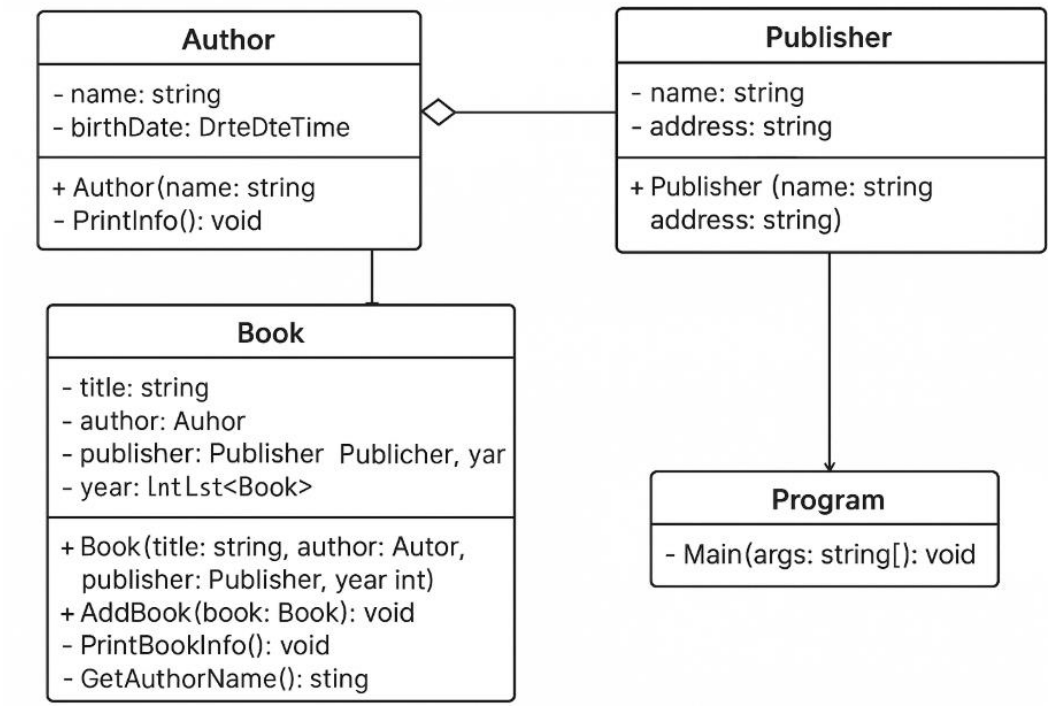
Створити класи «Author», «Publisher», «Book». Книга включає автора і видавництво. Використати композицію , реалізувати методи

- AddBook (Book t) – додати завдання
- PrintBookInfo() – вивід інформації

- GetAuthorName() – отримання автора

Використовувати геттери-сеттери, інкапсуляцію, конструктори.

Діаграма класів:



Програмна реалізація завдання:

```
//Publisher
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace OOP_9
{
    class Publisher
    {
        private string name;
        private string address;
        public Publisher(string name, string address)
        {
            this.name = name;
```

```
            this.address = address;
        }
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
        public string Address
        {
            get { return address; }
            set { address = value; }
        }
        public void PrintInfo()
        {
            Console.WriteLine($"Publisher: {name}, Address: {address}");
        }    }
}
//Books
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace OOP_9
{
    class Book
    {
        private string title;
        private Author author;
        private Publisher publisher;
        private int year;
        private List<Book> books;
        public Book(string title, Author author, Publisher publisher, int year)
        {
```

```
        this.title = title;
        this.author = author;
        this.publisher = publisher;
        this.year = year;
        this.books = new List<Book>();
    }
    public string Title
    {
        get { return title; }
        set { title = value; }
    }
    public Author Author
    {
        get { return author; }
        set { author = value; }
    }
    public Publisher Publisher
    {
        get { return publisher; }
        set { publisher = value; }
    }
    public int Year
    {
        get { return year; }
        set { year = value; }
    }
    public void AddBook(Book book)
    {
        books.Add(book);
        Console.WriteLine($"Book '{book.Title}' added to the collection.");
    }
    public void PrintBookInfo()
    {
        Console.WriteLine($"Book Title: {title}");
```

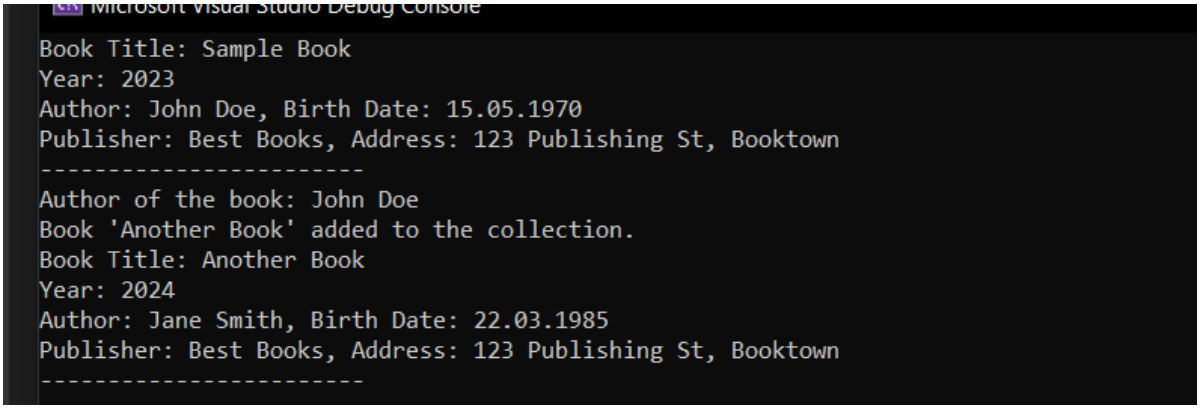
```
            Console.WriteLine($"Year: {year}");
            author.PrintInfo();
            publisher.PrintInfo();
            Console.WriteLine("-----------------------");
        }
        public string GetAuthorName()
        {
            return author.Name;
        }
    }
}

//Author
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace OOP_9
{
    using System;
    using System.Collections.Generic;

    class Author
    {
        private string name;
        private DateTime birthDate;
        public Author(string name, DateTime birthDate)
        {
            this.name = name;
            this.birthDate = birthDate;
        }
        public string Name
        {
            get { return name; }
```

```
            set { name = value; }
        }
        public DateTime BirthDate
        {
            get { return birthDate; }
            set { birthDate = value; }
        }
        public void PrintInfo()
        {
            Console.WriteLine($"Author: {name}, Birth Date:
{birthDate.ToShortDateString()}");
        }
    }
}
//Program
using OOP_9;
class Program
{
    static void Main(string[] args)
    {
        Author author = new Author("John Doe", new DateTime(1970, 5, 15));
        Publisher publisher = new Publisher("Best Books", "123 Publishing St,
Booktown");
        Book book1 = new Book("Sample Book", author, publisher, 2023);
        book1.PrintBookInfo();
        Console.WriteLine($"Author of the book: {book1.GetAuthorName()}");
        Book book2 = new Book("Another Book", new Author("Jane Smith", new
DateTime(1985, 3, 22)), publisher, 2024);
        book1.AddBook(book2);
        book2.PrintBookInfo();
    }
}
```

Результат виконання програми:



Посилання на репозиторій з проєктом: …

Висновки:

У ході виконання лабораторної роботи було реалізовано систему класів Author, Publisher та Book з використанням принципів композиції, де клас Book включає об'єкти автор та видавництво. Впроваджено методи AddBook(), PrintBookInfo() та GetAuthorName() із застосуванням інкапсуляції через геттери-сеттери та конструктори для правильної ініціалізації об'єктів.

МАТЕРІАЛ ДЛЯ САМОСТІЙНОГО ОПРАЦЮВАННЯ

Самостійна робота студента з дисципліни включає: опрацювання теоретичного матеріалу (в тому числі й опрацювання окремих тем програми або їх частини, які не викладаються на аудиторних заняттях); підготовку до виконання і захисту лабораторних робіт, підготовку до контрольних робіт та підсумкового екзамену.

Матеріал тем, винесених на самостійне опрацювання, включений до теоретичних питань на контрольних роботах та до екзаменаційних питань. Частина матеріалу обговорюється при виконанні та захисті лабораторних робіт або у секції відповіді на запитання студентів під час лекції.

 ***Питання, що виносяться на самостійне опрацювання:***

<u>1 СЕМЕСТР</u>

- UML-діаграми у проектуванні програмних систем.
- Роль абстракції в об’єктно-орієнтованому моделюванні
- Дружні функції та їх застосування у C++.
- Делегуючі конструктори та їх використання у сучасному C++.
- Правило «Великої трійки» та «Великої п’ятірки» у C++.
- Перевантаження операторів: доцільність і типові помилки при використанні.
- Статичні члени класу: сфери застосування, переваги та обмеження.
- Множинне успадкування та проблема «ромбовидного» успадкування.
- Абстрактні класи й інтерфейси: відмінності та сфери застосування.
- Механізми обробки виняткових ситуацій у C++: створення власних винятків.
- Основи роботи з контейнерами та алгоритмами STL у C++.
- Модульність та інкапсуляція у великих програмах.

- Передача об’єктів у функції: за значенням, за посиланням, за вказівником.
- Використання UML-діаграм для проєктування класів і взаємозв’язків.

2 СЕМЕСТР

- Використання систем контролю версій (Git) у командній ООП-розробці.
- Сучасні парадигми програмування та їх поєднання: ООП, функціональне програмування, реактивне програмування.
- SOLID-принципи в ООП: приклади реалізації на сучасних мовах програмування.
- Патерни проєктування (GoF, GRASP): застосування в реальних проєктах.
- Unit-тестування об’єктно-орієнтованих програм (Google Test, JUnit, NUnit).
- Інструменти автоматизації збірки та управління залежностями (CMake, Maven, Gradle).
- Інкапсуляція та модульність у великих проєктах: принципи та приклади.
- Інтерфейси та абстрактні класи у проєктуванні API.
- Асинхронне та багатопотокове програмування в ООП.
- Робота з бібліотеками та фреймворками у контексті ООП (Qt, Spring, .NET).
- Використання шаблонів (generics/templates) для узагальненого програмування.
- Рефакторинг ООП-коду: техніки та інструменти (SonarQube, ReSharper).
- Впровадження інверсії керування (IoC) та впровадження залежностей (DI).
- Організація тестового покриття у великих ООП-системах.
- Безпека ООП-додатків: захист від SQL-ін’єкцій, XSS, буферних переповнень.
- Використання контейнеризації (Docker) для розгортання ООП-додатків.

- Мікросервісна архітектура у контексті ООП.
- Сучасні IDE для ООП-розробки та їхні можливості.

 ***Запитання для самоконтролю за матеріалом, винесеним на самостійне опрацювання***

1. Яка з систем контролю версій найчастіше використовується у командній розробці?

a) Subversion (SVN)

b) Git

c) CVS

d) Mercurial

2. Яка з парадигм програмування базується на використанні чистих функцій та відсутності побічних ефектів?

a) Об’єктно-орієнтоване програмування

b) Функціональне програмування

c) Реактивне програмування

d) Імперативне програмування

3. Який принцип SOLID означає, що клас повинен мати лише одну відповідальність?

a) Open/Closed Principle

b) Single Responsibility Principle

c) Dependency Inversion Principle

d) Interface Segregation Principle

4. Який патерн відноситься до структурних (за класифікацією GoF)?

a) Singleton

b) Adapter

c) Observer

d) Strategy

5. Який інструмент використовується для unit-тестування в C++?

a) JUnit

b) NUnit

c) Google Test

d) Jasmine

6. Який інструмент зазвичай використовується для автоматизації збірки у C++-проектах?

a) Maven

b) Gradle

c) CMake

d) Ant

7. Яке твердження найточніше описує абстрактний клас у проєктуванні API?

a) Може мати лише абстрактні методи

b) Може містити як реалізовані, так і абстрактні методи

c) Не може мати конструкторів

d) Не підтримує наслідування

8. Яка технологія забезпечує ізольоване розгортання ООП-додатків у контейнерах?

a) VirtualBox

b) Docker

c) VMWare

d) Kubernetes

9. Який принцип використовується для впровадження залежностей у великі системи?

a) DRY

b) IoC

c) KISS

d) YAGNI

10. Яка можливість сучасних IDE (наприклад, Rider, Visual Studio, IntelliJ IDEA) суттєво прискорює роботу з ООП-кодом?

a) Виконання лише у терміналі

b) Автоматична генерація коду та рефакторинг

c) Підтримка тільки однієї мови програмування

d) Відсутність інтеграції з Git

11. Який із перелічених фреймворків належить до C#/.NET і широко використовується для побудови корпоративних застосунків?

a) Qt

b) Spring

c) Boost

d) ASP.NET Core

12. У C# при оголошенні класу, що успадковує інший клас і реалізує інтерфейс, використовується синтаксис:

a) class Child : Base, IInterface

b) class Child implements Base, IInterface

c) class Child extends Base, IInterface

d) class Child -> Base, IInterface

13. Яке з наведених тверджень найкраще описує призначення інструментів SonarQube та ReSharper у контексті ООП-розробки?

a) Використовуються для компіляції та запуску програм

b) Виконують статичний аналіз коду, виявляють помилки та рекомендують рефакторинг

c) Призначені лише для управління версіями коду

d) Автоматизують процес збірки та розгортання застосунку

14. Що з поданого найточніше описує роль інкапсуляції та модульності у великих ООП-проєктах?

a) Забезпечують приховування внутрішньої реалізації та полегшують супровід коду

b) Використовуються лише для оптимізації продуктивності

c) Зменшують розмір виконуваних файлів

d) Є синонімами до успадкування та поліморфізму

15. У чому полягає ключова відмінність між інтерфейсом та абстрактним класом при проєктуванні API?

a) Абстрактний клас може містити реалізацію методів, а інтерфейс - ні

b) Інтерфейс завжди працює швидше за абстрактний клас

c) Інтерфейси створюються лише у C#, а абстрактні класи – у C++

d) Абстрактні класи не підтримують успадкування

16. Яку роль відіграє Git у командній об’єктно-орієнтованій розробці?

17. Поясніть, у чому полягає відмінність між ООП та функціональним програмуванням.

18. Наведіть приклад реалізації Single Responsibility Principle (SRP) у програмі на C++.

19. Що таке патерн проєктування Adapter і в яких випадках його застосовують?

20. Для чого використовується Unit-тестування у процесі розробки ООП-програм?

21. Яке призначення інструмента CMake у великих проєктах на C++?

22. У чому різниця між абстрактним класом та інтерфейсом при проєктуванні API?

23. Поясніть, як Docker може допомогти при розгортанні ООП-додатків.

24. Що таке інверсія керування (IoC) та яка її користь для масштабних проєктів?

25. Які можливості IDE (наприклад, Visual Studio чи JetBrains Rider) спрощують рефакторинг та налагодження ООП-програм?

26. Встановіть відповідність між SOLID-принципом та його описом::

A) Single Responsibility Principle (SRP) B) Open/Closed Principle (OCP) C) Liskov Substitution Principle (LSP) D) Dependency Inversion Principle (DIP)	1) Модулі вищого рівня не повинні залежати від модулів нижчого рівня. 2) Клас повинен мати лише одну причину для зміни. 3) Програмні сутності повинні бути відкритими для розширення, але закритими для модифікації 4) Об'єкти суперкласу повинні замінюватися об'єктами підкласів без порушення функціональності.

27. Встановіть відповідність між патерном проєктування та його призначенням:

A) Adapter B) Observer C) Factory Method D) Singleton	1) Дозволяє створювати об'єкти без вказування точного класу 2) Забезпечує сумісність між класами з несумісними інтерфейсами 3) Визначає залежність "один-до-багатьох" між об'єктами 4) Гарантує, що клас має лише один екземпляр

28. Встановіть відповідність між інструментом та його призначенням:

A) Git B) CMake C) Maven D) JUnit	1) Фреймворк для unit-тестування Java-додатків 2) Система контролю версій для відстеження змін у коді 3) Кросплатформний інструмент для автоматизації збірки C++ проєктів 4) Система управління залежностями та збірки для Java проєктів

29. Встановіть відповідність між концепцією generics у C# та її описом:

A) Generic Class B) Type Constraint C) Generic Method D) Type Parameter (T)	1) Обмеження типу для забезпечення певної функціональності (where T : IComparable) 2) Клас, що може працювати з різними типами (List<T>, Dictionary<K,V>) 3) Символ, що використовується для позначення типу-параметра 4) Метод, що може приймати параметри різних типів (<T> T GetValue())

30. Встановіть відповідність між Git-командою/концепцією та її призначенням у контексті ООП-розробки:

A) git merge B) git branch feature/user-service C) git pull request/merge request D) git rebase	1) Створення окремої гілки для розробки нового класу чи модуля 2) Об'єднання змін з різних гілок після завершення розробки компонентів 3) Механізм перегляду коду перед інтеграцією нових класів в основну гілку 4) Перенесення комітів з однієї гілки на іншу для чистої історії розробки

 ***Відповіді на запитання для самоконтролю за матеріалом, винесеним на самостійне опрацювання***

1. B) Git
2. B) Функціональне програмування
3. B) Single Responsibility Principle
4. B) Adapter
5. C) Google Test
6. C) CMake
7. B) Може містити як реалізовані, так і абстрактні методи
8. B) Docker
9. B) IoC
10. B) Автоматична генерація коду та рефакторинг
11. D) ASP.NET Core
12. A) class Child : Base, IInterface
13. B) Виконують статичний аналіз коду, виявляють помилки та рекомендують рефакторинг

14. А) Забезпечують приховування внутрішньої реалізації та полегшують супровід коду

15. А) Абстрактний клас може містити реалізацію методів, а інтерфейс – ні

16. Git забезпечує версійний контроль коду, дозволяючи декільком розробникам одночасно працювати над різними класами та модулями без конфліктів. Він надає можливість відслідковувати зміни в структурі класів, відкочувати помилкові рефакторинги та об'єднувати код з різних гілок розробки.

17. ООП організовує код навколо об'єктів (даних та методів), підтримуючи інкапсуляцію, наслідування та поліморфізм. Функціональне програмування базується на функціях як основних будівельних блоках, уникає зміни стану та побічних ефектів, використовуючи незмінні дані та функції вищого порядку.

18. Можна навести наступний приклад:

```
// Відповідність SRP
class PayrollCalculator {
double calculatePay(const Employee& emp) { /* розрахунок */ }
};
class EmployeeRepository {
void save(const Employee& emp) { /* збереження */ }
};
```

19. Adapter дозволяє об'єктам з несумісними інтерфейсами працювати разом, створюючи проміжний клас-адаптер. Використовується при інтеграції сторонніх бібліотек, легаси-коду або коли потрібно змінити інтерфейс існуючого класу без його модифікації.

20. Unit-тестування перевіряє коректність роботи окремих класів та методів у ізоляції від інших компонентів. Воно допомагає виявляти помилки на ранніх стадіях розробки та забезпечує можливість безпечного рефакторингу без порушення функціональності.

21. CMake автоматизує процес збірки проєктів, управляє залежностями між модулями та генерує файли для різних систем збірки (Make, Visual Studio, Ninja). Він забезпечує кросплатформну сумісність та спрощує конфігурацію складних проєктів з множиною бібліотек.

22. Абстрактний клас може містити як абстрактні методи, так і конкретну реалізацію, підтримує наслідування стану та конструктори. Інтерфейс (або чисто абстрактний клас у C++) містить лише сигнатури методів без реалізації, забезпечуючи контракт поведінки без зв'язування з конкретною реалізацією.

23. Docker упаковує додаток та всі його залежності в контейнер, забезпечуючи однакове середовище виконання на різних платформах. Він спрощує розгортання складних ООП-систем з множиною сервісів, забезпечує ізоляцію компонентів та полегшує масштабування мікросервісної архітектури.

24. IoC - це принцип, при якому об'єкт не створює свої залежності самостійно, а отримує їх ззовні (через конструктор, сетери або DI-контейнер). Це зменшує зв'язаність між класами, полегшує тестування через можливість підміни залежностей та робить код більш гнучким для змін.

25. IDE надають автоматичне перейменування класів та методів з оновленням всіх посилань, інтелектуальне автодоповнення та навігацію по коду. Для налагодження доступні точки зупинки з умовами, покрокове виконання, інспекція об'єктів та їх стану, а також візуалізація стеку викликів та ієрархії класів.

26. A-2, B-3, C-4, D-1

27. A-2, B-3, C-1, D-4

28. A-2, B-3, C-4, D-1

29. A-2, B-1, C-4, D-3

30. A-2, B-1, C-3, D-4

ЗАСОБИ ДЛЯ ПРОВЕДЕННЯ ПОТОЧНОГО ТА ПІДСУМКОВОГО КОНТРОЛЮ

Згідно з навчальним планом, дисципліна вивчається два семестри і формою підсумкового контролю досягнутих успіхів студента у кожному семестрі є екзамен. Досягнуті успіхи студента з дисципліни оцінюються під час виконання та захисту лабораторних робіт, контрольними роботами, під час захисту індивідуального завдання та екзаменом.

Протягом першого семестру пропонується виконати 13 *лабораторних робіт*, протягом другого – 9 *лабораторних робіт*, на кожній із яких виконуються індивідуальні завдання. До захисту необхідно опрацювати поданий у методичних вказівках теоретичний матеріал. За виконання лабораторних завдань можна отримати максимум 50 балів щосеместру.

Кількість балів, що виставляється за лабораторне заняття, враховує: знання теоретичного матеріалу з теми; повноту виконання поставлених завдань з теми; своєчасне виконання та захист лабораторної роботи. Термін захисту лабораторної роботи вважається своєчасним, якщо студент захистив її згідно з графіком. У разі не дотримання термінів захисту лабораторної роботи максимальна кількість балів за роботу зменшується на 1 бал щотижня. Студент, який навчається за тимчасовим індивідуальним навчальним планом, зобов'язані здати усі лабораторні роботи у терміни, зазначені у тимчасовому індивідуальному навчальному плані, і отримати певну суму балів.

Контрольні роботи передбачають виконання різнотипних тестових та практичних завдань. Під час виконання завдань контрольної роботи не можна користуватись ніякими джерелами. Контрольна робота проводиться в письмовій формі тестова частина (електронне тестування) оцінюється автоматично, практична частина оформляється відповідно до вимог та подається на перевірку..

Студенти повинні бути готові до усного захисту контрольної роботи (впродовж тижня з часу написання роботи). У результаті захисту оцінка може бути скорегована (в межах балів, передбачених для даної роботи). Студенти, які навчаються за тимчасовим індивідуальним планом, зобов'язані проходити контроль у терміни, визначені графіком.

За виконання контрольних робіт студент може отримати до 40 балів. Підсумковий екзамен проводиться в усній формі за екзаменаційними білетами, що передбачають виконання теоретичних та практичних завдань; містить завдання на 100 балів.

У першому семестрі передбачена співбесіда з лектором, що проводиться в усній формі протягом останнього навчального тижня. Дана форма контролю спрямована на перевірку здатності студента логічно, послідовно та аргументовано викладати навчальний матеріал; аналізувати та систематизувати інформацію тощо. Співбесіда також має на меті функцію самоперевірки рівня власних знань студентом та визначення слабких місць перед підготовкою до іспиту.

Максимальну за результатами співбесіди студент може отримати 10 балів. Критерії оцінювання співбесіди:

- високий рівень (9 –10 балів) – студент повно і послідовно розкриває питання, демонструє глибоке розуміння теоретичних положень, дає розгорнуті відповіді без істотних неточностей;
- достатній рівень (7 – 8 балів) – відповіді у цілому правильні й аргументовані, але присутні незначні неточності чи неповнота викладу; приклади з практики наведені частково;
- середній рівень (5 – 6 балів) – студент відтворює основний зміст питань, проте відповідь поверхнева, малоструктурована, з помітними прогалинами у розумінні; приклади практичного застосування відсутні або нечіткі;

- низький рівень (3 – 4 бали): відповіді неповні, уривчасті, з численними неточностями та труднощами у використанні термінології;
- дуже низький рівень (0 – 2 бали): студент не розкриває суті поставлених питань, відповіді фрагментарні або неправильні, відсутнє розуміння базових понять.

У другому семестрі планується виконання навчального проєкту з використанням технологій ООП та командної роботи, за виконання якого студенти можуть отримати до 15 балів максимально. Дана форма контролю має на меті поглиблення, узагальнення та закріплення знань, які студенти отримують у процесі навчання, а також застосування цих знань на практиці.

Індивідуальне завдання передбачає розроблення застосунку з використання технологій ООП та командного підходу до реалізації ІТ-проєктів. Воно обов'язково передбачає теоретичну частину та практичну (розроблення додатку для демонстрації практичного застосування опрацьованого матеріалу) частини; оформляється у письмовій формі (з потрібними додатками) і подається на перевірку не пізніше, ніж за тиждень до завершення семестру.

Критерії оцінювання індивідуальних завдань:

Критерії	**Бали**
Повнота та логічність матеріалу, висвітленого в завданні, відповідність поставленим вимогам та темі проєкту.	до 5 балів
Організація командної роботи на проєктом	до 3 балів
Якість усного виступу під час захисту завдання, доступність та чіткість представлення матеріалу, використання допоміжних засобів, відповідь на поставлені запитання.	до 3 балів
Дотримання вимог до оформлення звітних матеріалів.	до 2 балів
Дотримання термінів подання завдання на перевірку лектору.	до 2 балів

Індивідуальне завдання потребує усного захисту у формі доповіді перед всіма студентами групи з подальшим обговоренням проблемних питань, розкритих в завданні.

Студенти можуть самостійно обрати тему індивідуального завдання, попередньо узгодивши її з лектором.

Сумарна кількість балів з дисципліни за семестр визначається як поточна успішність (сума балів з усіх видів навчальної роботи). Оцінка виставляється за шкалами оцінювання: стобальною, національною і ЄКТС. Семестрова підсумкова оцінка визначається як сума балів з усіх видів навчальної роботи

1 семестр

Види робіт:	**Розподіл**	**Екзамен**
Контрольні роботи	40	
Захист лабораторних робіт	50	
Співбесіда з лектором	10	
Всього балів	100	100
Ваговий коефіцієнт	0,6	0,4

2 семестр

Види робіт:	**Розподіл**	**Екзамен**
Контрольні роботи	35	
Захист лабораторних робіт	50	
Виконання навчального проєкту	15	
Всього балів	100	100
Ваговий коефіцієнт	0,6	0,4

Оцінювання знань з дисципліни здійснюється на основі результатів поточного контролю знань за 100-бальною шкалою, після чого переводиться в національну шкалу оцінювання та шкалу ECTS відповідно до наступних критеріїв:

$$\mathbf{S_{сум} = 0.6* S_{пот}+0.4* S_{підс}}$$

де $\mathbf{S_{пот}}$ – сума балів, отриманих за лабораторні та контрольні роботи й виконання індивідуального завдання, $\mathbf{S_{підс}}$ – сума балів, отриманих на екзамені за відповіді на питання та виконання завдань екзаменаційного білету.

Екзамен за талоном №2 і перед комісією проводиться в письмово-усній формі з оцінюванням за стобальною шкалою. Якщо студент вимушений складати екзамен через незадовільну оцінку за результатами поточного контролю (Т. 2 чи Т. К), то оцінка на екзамені виставлятиметься без урахування балів, набраних під час поточного семестрового контролю.

ВИКОНАННЯ ІНДИВІДУАЛЬНОГО ЗАВДАННЯ

В ході вивчення курсу студенти мають в якості однієї з форм контролю рівня навчальних досягнень виконання індивідуальних завдання, які мають форму командних проєктів та передбачають практичну роботу над проблемними питаннями з тематики дисципліни. Індивідуальні завдання мають на меті поглиблення, узагальнення та закріплення знань, які студенти отримують у процесі навчання, а також застосування цих знань на практиці.

Індивідуальне завдання для курсу ООП передбачає розроблення програмного застосунку з використання технологій ООП та командного підходу до реалізації ІТ-проєктів.

Індивідуальне навчально-дослідне завдання обов’язково передбачає теоретичну частину та практичну (розроблення додатку для демонстрації практичного застосування опрацьованого матеріалу) частини; оформляється у письмовій формі (з потрібними додатками) і подається лектору не пізніше, ніж за тиждень до завершення семестру. Індивідуальне завдання потребує усного захисту у формі доповіді перед всіма студентами групи з подальшим обговоренням проблемних питань, розкритих в завданні.

Студенти може самостійно обрати тему індивідуального завдання, попередньо узгодивши її з лектором.

Командна робота студентів над розробленням програмних продуктів дозволяє не лише глибше осягнути принципи програмної інженерії, а й розвинути ключові «м’які навички» – відповідальність, уміння ефективно спілкуватися та координувати роботу в команді. Щоб оволодіння розробкою програм було успішним, теоретичні знання про об’єктну модель необхідно поєднувати з практичним використанням концепцій моделювання в умовах, максимально наближених до реальних завдань командної роботи над програмним забезпеченням.

Завдання передбачає створення програмного застосунку мовою C# (наприклад, чат, планувальник задач, конвертер або мінігра), у якому студенти мають практично реалізувати принципи об’єктно-орієнтованого моделювання в умовах спільної роботи команди. Організація діяльності та зміст завдання спрямовані на імітацію реальних процесів розробки програмного забезпечення. На відміну від індивідуальних лабораторних завдань, цей формат забезпечує можливість колективного створення програмного продукту (зазвичай, з використанням хмарних середовищ для організації спільної роботи над написанням коду), поєднуючи застосування принципів ООП із розвитком навичок командної взаємодії.

Для виконання проєкту студенти мають об’єднатись у групи по 3–4 особи, а всі етапи – від постановки задачі до презентації результатів – реалізуються із використанням сучасних хмарних сервісів та документуються.

Обов’язкові вимоги до виконання завдання є наступними:

– розробити проєкт засобами мови C#, спроєктувавши певну архітектуру класів. Мінімальна вимога – використання не менше п’яти класів, між якими мають бути реалізовані зв’язки (успадкування, композиція, асоціації тощо). Проєкт повинен становити завершений програмний продукт, який має логічне призначення та демонструє практичне застосування принципів об’єктно-орієнтованого програмування.

– результат завантажити до публічного репозиторію GitHub або Replit.

– оформити звіт про виконання завдання, що містить постановку завдання, призначення проєкту, обрані засоби розроблення, UML-діаграму класів, скриншоти роботи застосунку з поясненнями його функціональності, опис ролей і внеску кожного учасника команди та посилання на репозиторій з проєктом.

Проєкт потребує усного захисту у формі доповіді перед всіма студентами групи з подальшим обговоренням проблемних питань, розкритих в завданні. Для

цього команда має заздалегідь продумати структуру презентації проєкту, підготувавши відповідні матеріали (слайди з UML-діаграмами, скриншотами роботи програми, описом архітектури та ролей учасників). Крім того, студенти повинні бути готові не лише представити результати своєї роботи, а й надати аргументовані відповіді на запитання слухачів і викладача під час захисту, що сприятиме кращому розумінню сутності виконаного завдання та підтвердить глибину його опрацювання.

Усі етапи виконання завдання – від вибору теми й постановки задачі до тестування та підготовки до захисту – рекомендовано обговорювати з лектором. Для цього організовуються консультації, де студенти можуть уточнити вимоги, отримати рекомендації щодо архітектури проєкту, використання інструментів чи подолання труднощів у роботі команди

Для забезпечення якісної командної взаємодії студенти мають застосовувати сучасні засоби організації спільної роботи:

• системи контролю версій (Git);

• хмарні середовища для розробки (Replit, GitHub Codespaces, Visual Studio Live Share);

• інструменти для комунікації та планування (Slack, Microsoft Teams, Trello, Notion тощо).

Особливу увагу слід приділити документуванню командної роботи – фіксації рішень, протоколюванню обговорень, збереженню технічних завдань та опису змін у коді через систему комітів. Це дозволить прозоро розподіляти відповідальність між учасниками і спростить захист проєкту.

Рекомендовані хмарні сервіси для роботи над проєктом

Для підвищення ефективності роботи над індивідуальним командним завданням доцільно використовувати сучасні хмарні сервіси, що підтримують командну взаємодію, контроль версій і документування:

GitHub

GitHub – платформа для зберігання коду, спільної роботи над ним та використання системи контролю версій Git. Забезпечує можливість створення публічних та приватних репозиторіїв, налаштування прав доступу, а також інтеграцію з іншими інструментами CI/CD.

Використання GitHub дасть змогу:

- ефективно відслідковувати всі зміни в коді;
- створювати гілки для розробки окремих функцій або модулів;
- фіксувати стабільні версії додатку (release points);
- автоматизувати процеси безперервної інтеграції;
- вести документацію безпосередньо в репозиторії.

На рис. 1 наведено вигляд приватного репозиторію GitHub, створеного для роботи з даними.

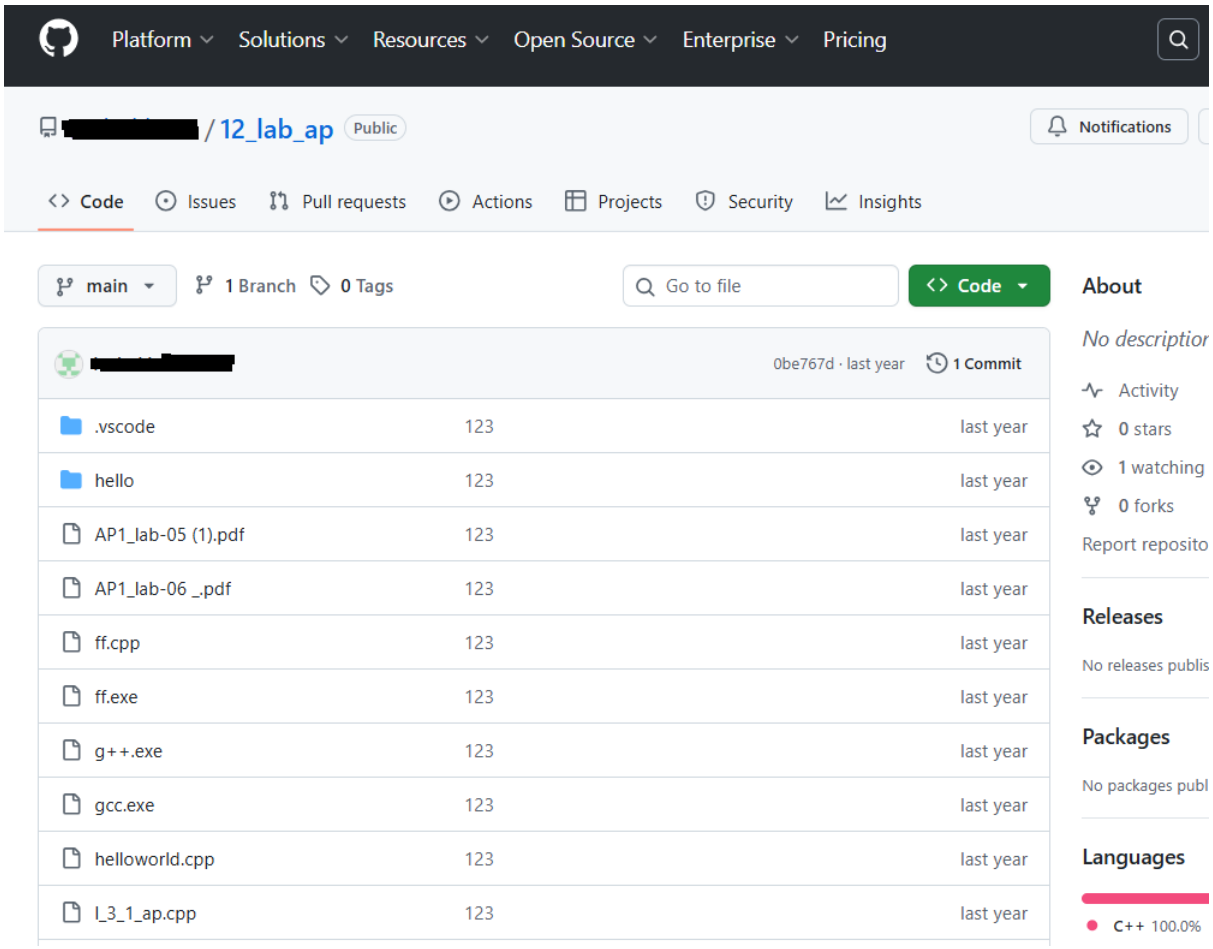


Рис. 1. Репозиторій проєкту на GitHub

Вибір GitHub обумовлений також широкою інтеграцією з іншими сервісами, зручним веб-інтерфейсом та підтримкою відкритого доступу або приватного репозиторію, що актуально для роботи над проєктом. Завдяки використанню GitHub, розробка додатку стає більш керованою, прозорою та надійною з погляду резервного копіювання та історії змін.

Replit

Replit – інтерактивне середовище розробки в браузері, яке дозволяє писати, запускати й тестувати код у режимі реального часу. Зручне для командної роботи, оскільки кілька користувачів можуть одночасно працювати над одним проєктом. Це середовище підтримує понад 50 мов програмування, включаючи C++ та C#, і надає можливість працювати з проєктами як індивідуально, так і в команді.

Для початку роботи у Replit достатньо зареєструватися на платформі та створити новий проєкт (repl), обравши C++ (або C#) як мову програмування. Інтерфейс Replit складається з трьох основних частин: зліва розташований файловий менеджер, де можна створювати та керувати файлами проєкту; у центрі знаходиться редактор коду з підтримкою підсвічування синтаксису та автодоповненням коду; справа – консоль для виведення результатів роботи програми та термінал для виконання команд (рис. 2).

Replit інтегрував власний ШІ-інструмент, який суттєво розширює можливості середовища розробки.

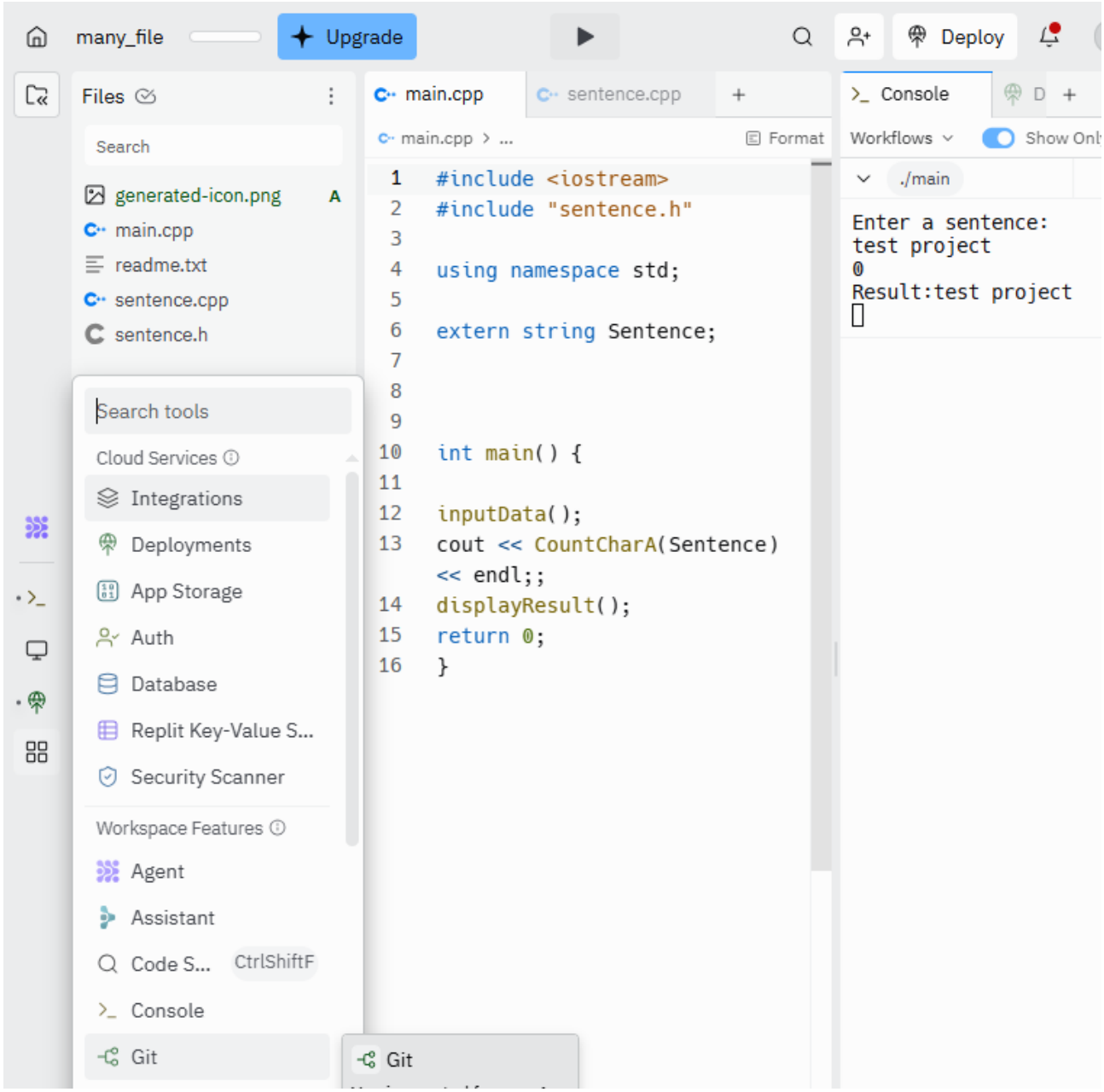


Рис. 2. Виконання програми у Replit

Agent&Assistant працює як інтелектуальний помічник програміста, що може допомагати з написанням, аналізом та оптимізацією коду.

Він пропонує такі функції як автодоповнення коду, генерація документації, пояснення складних частин коду та навіть допомога у відлагодженні (рис. 3).

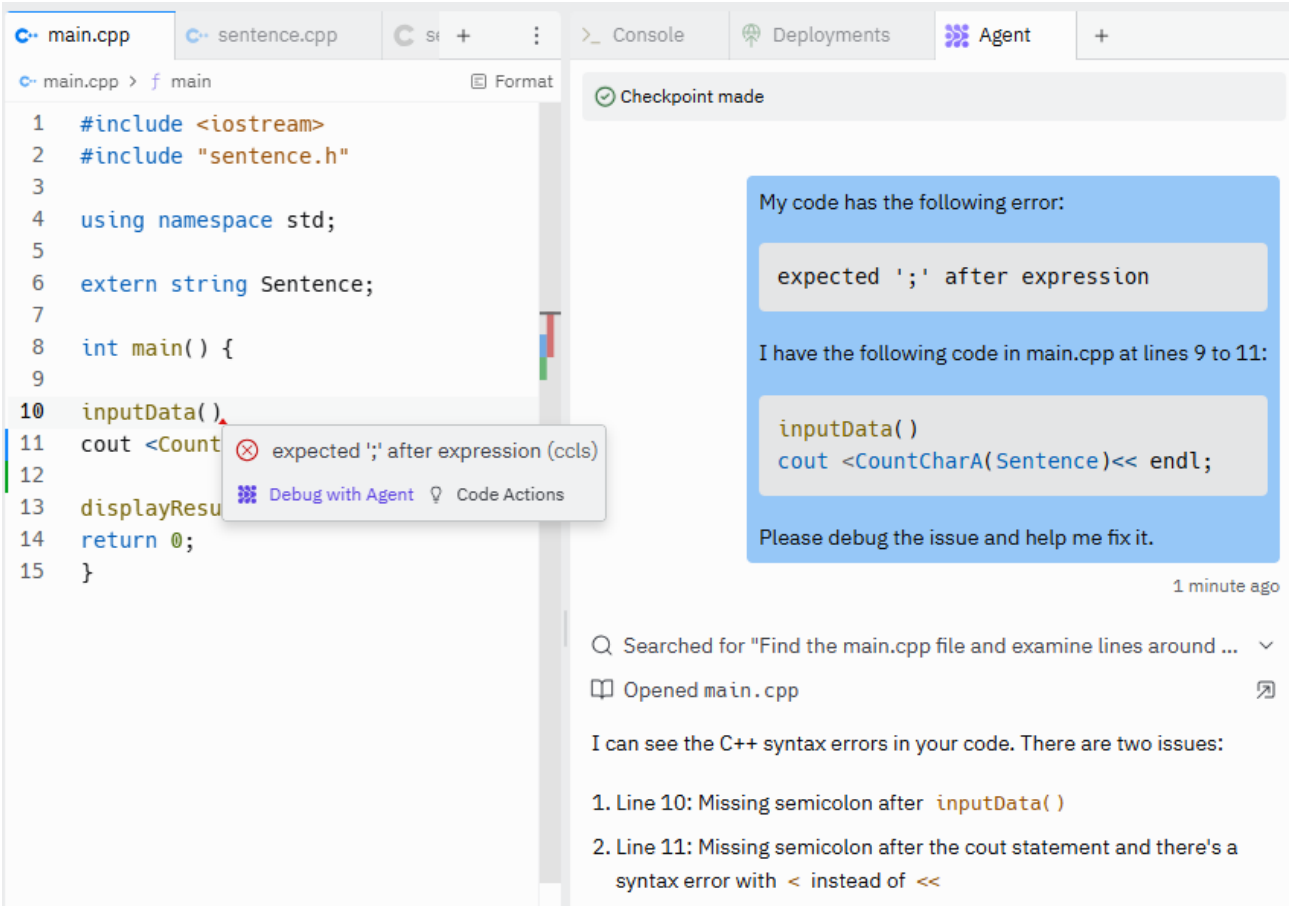


Рис. 3. Використання ШІ помічника у Replit

Agent спеціалізується на допомозі з кодом та може виконувати складні завдання з розробки. Він здатний:

- Аналізувати існуючий код та пропонувати оптимізації
- Генерувати нові функції та модулі на основі опису функціональності
- Допомагати з рефакторингом коду
- Створювати модульні тести
- Знаходити та виправляти помилки у коді
- Пояснювати складні алгоритми та структури даних

Assistant Replit виступає як інтерактивний помічник, який може:

- Відповідати на питання про програмування та допомагати з вирішенням проблем
- Надавати покрокові пояснення щодо роботи коду

- Пропонувати навчальні матеріали та приклади
- Допомагати з налаштуванням середовища розробки
- Надавати рекомендації щодо кращих практик програмування
- Генерувати документацію для коду

Особливістю цих інструментів є їхня глибока інтеграція з середовищем Replit - вони мають доступ до всього контексту проєкту, включаючи історію змін, залежності та конфігурації. Це дозволяє надавати більш точні та контекстно-релевантні підказки та рекомендації.

Під час написання програм на C++ розробник може використовувати ШІ для отримання контекстних підказок та рекомендацій щодо покращення коду. Наприклад, при роботі зі складними структурами даних чи алгоритмами, ШІ може запропонувати більш ефективні рішення або вказати на потенційні проблеми в коді.

GitHub Codespaces

GitHub Codespaces – це хмарне середовище розробки, яке дозволяє створювати повноцінні робочі простори для програмування прямо в браузері або через VS Code. Кожен Codespace є віртуальною машиною з попередньо налаштованим оточенням для конкретного проекту. Сервіс забезпечує миттєвий запуск робочого середовища за лічені секунди та автоматичне налаштування через використання dev containers для попереднього конфігурування.

Основні переваги сервісу:

- Миттєвий запуск – створення робочого середовища за лічені секунди;
- автоматичне налаштування;
- пряма робота з GitHub-репозиторіями, pull requests та issues;
- доступ з будь-якого пристрою через браузер ;
- можливість вибору різних конфігурацій машин

- підтримка широкого спектру мов програмування завдяки попередньо налаштованим шаблонам для популярних стеків як Node.js, Python, Java, .NET, Go та інших.

Для командної роботи доступні функції Live Share для спільного редагування коду в реальному часі та seamless інтеграція з GitHub Flow для роботи з branches, PR та code review.

Безпека забезпечується через систему секретів та змінних для конфіденційної інформації, а повний контроль над середовищем досягається через доступ до терміналу. Можливість запуску та тестування веб-додатків через систему портів та сервісів робить Codespaces особливо корисним для швидкого онбордингу нових розробників, роботи з різних пристроїв та забезпечення однорідності середовища розробки в команді.

Notion

Notion – це універсальна платформа продуктивності, яка поєднує в собі можливості текстового редактора, бази даних, планувальника завдань та інструменту для командної співпраці. Цей сервіс дозволяє створювати структуровані робочі простори, де можна організувати всю інформацію проєкту в єдиному місці. Основою Notion є блочна система, де кожен елемент контенту представлений у вигляді окремого блоку, який можна легко переміщувати, редагувати та комбінувати з іншими елементами.

Для ведення документації Notion пропонує потужні можливості створення баз знань з ієрархічною структурою сторінок, підсторінок та посилань між ними. Можна створювати детальні технічні специфікації, користувацькі інструкції, FAQ розділи та довідкові матеріали з підтримкою різних типів контенту включаючи текст, зображення, відео, код-блоки та інтерактивні елементи.

Планування завдань в Notion реалізовано через гнучку систему баз даних, де можна створювати канбан-дошки, календарі, списки та галереї для візуалізації робочих процесів.

Командна взаємодія забезпечується через систему дозволів та ролей, коментарі в реальному часі, згадування учасників команди та інтеграцію з популярними сервісами як Slack, Google Calendar, Figma та GitHub. Можливість створення спільних робочих просторів дозволяє різним відділам та проєктним командам мати доступ до релевантної інформації з контрольованими правами редагування (рис. 4).

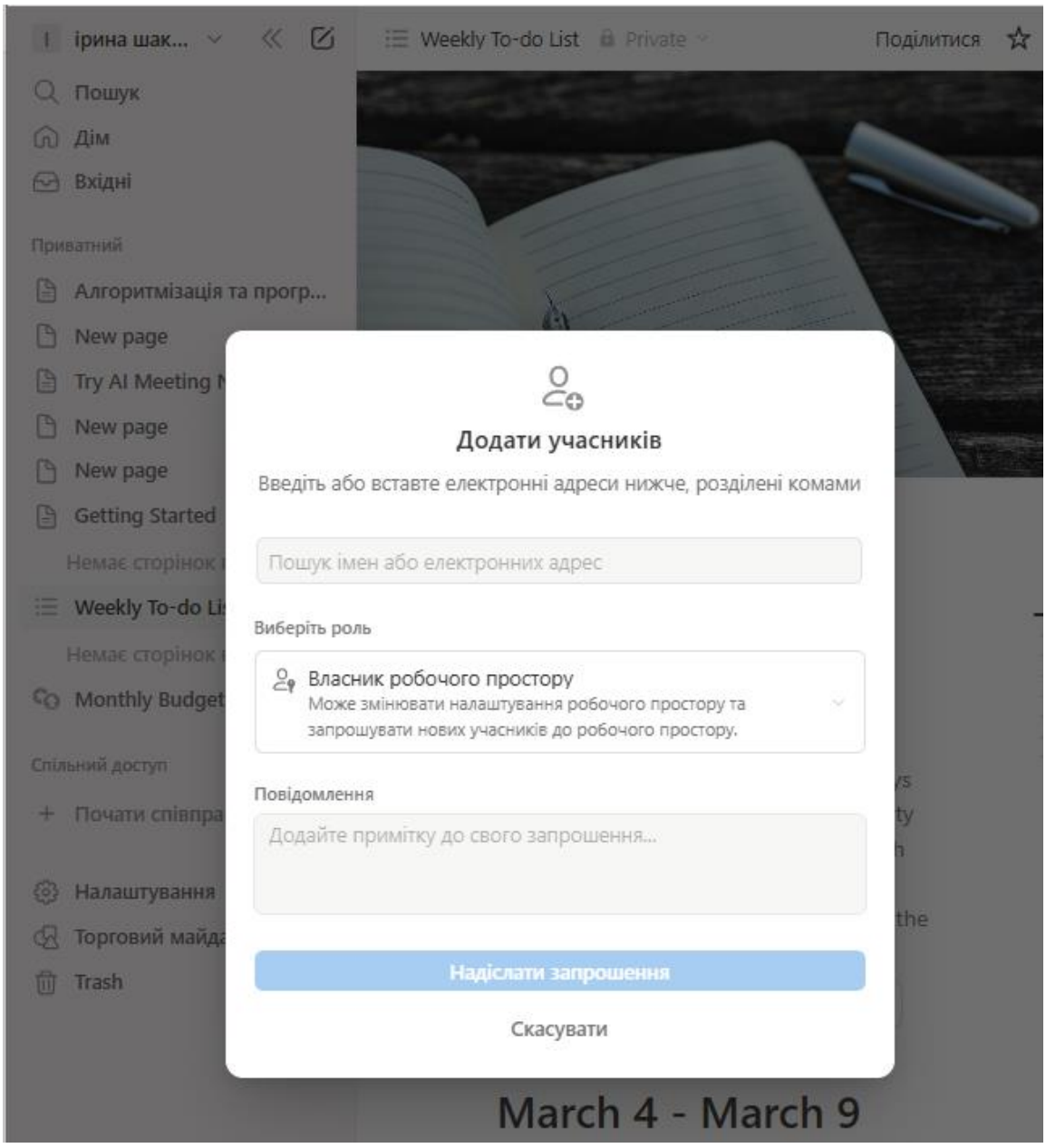


Рис. 4. Створення спільного робочого простору для команди у Notion

Trello або Microsoft Teams – сервіси для планування завдань і управління командною взаємодією. Trello зручний для візуалізації процесу роботи за допомогою канбан-дошок, тоді як Teams інтегрує комунікацію, календарі й документообіг.

Slack – платформа для командних комунікацій, що підтримує інтеграції з GitHub, Trello та іншими сервісами, дозволяючи швидко обмінюватися повідомленнями й отримувати сповіщення про зміни в проєкті.

При підготовці UML-діаграм класів, діаграм послідовності чи інших візуалізацій структури проєкту доцільно використовувати такі інструменти, як diagrams.net та Lucidchart.

diagrams.net

diagrams.net – безкоштовний вебсервіс для створення діаграм будь-якої складності. Інтегрується з Google Drive та GitHub, підтримує командну роботу та експорт у різні формати (рис. 5).

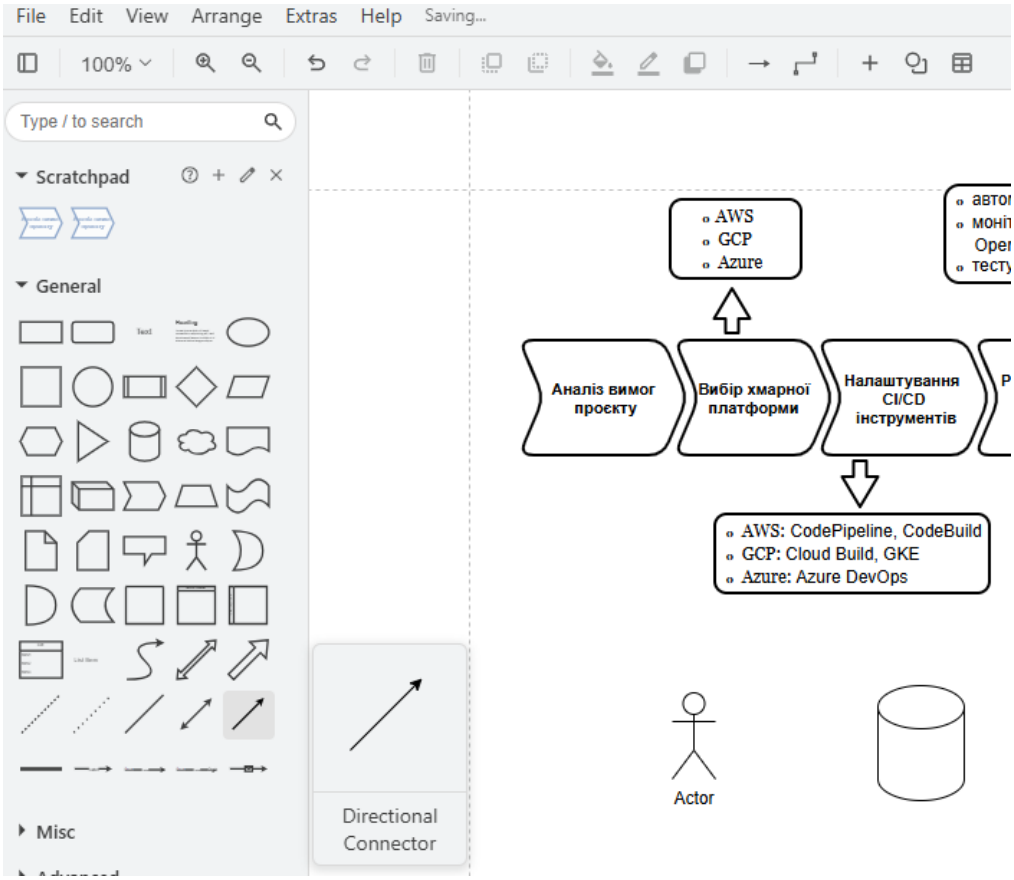


Рис. 5. Створення діаграм засобами diagrams.net

Командна робота в diagrams.net реалізована через систему спільного доступу та колаборації в реальному часі. Декілька користувачів можуть одночасно редагувати одну діаграму, бачити курсори інших учасників та миттєво отримувати оновлення змін. Система коментарів дозволяє залишати примітки до окремих елементів діаграми для обговорення деталей та внесення правок. Можливість налаштування різних рівнів доступу забезпечує контроль над тим, хто може переглядати, редагувати або коментувати діаграми.

Lucidchart

Lucidchart – професійний онлайн-редактор для побудови UML, ER-діаграм та блок-схем. Підтримує спільне редагування, коментування та інтеграцію з іншими сервісами (Google Workspace, Microsoft Teams, Slack).

Окремої уваги потребує етап захисту індивідуального завдання, під час якого студенти презентують результати роботи команди. Для створення якісних презентацій рекомендується скористатися такими сервісами:

Google Презентації (Google Slides) – безкоштовний хмарний сервіс, який забезпечує спільну роботу над презентацією в реальному часі, підтримує імпорт та експорт у різних форматах (PowerPoint, PDF) та інтеграцію з Google Drive. Платформа пропонує широкий набір професійних шаблонів та тем, які можна легко адаптувати під специфіку проєкту, а також підтримує вставку інтерактивних елементів як відео з YouTube, зображення з Google Photos та діаграми з Google Sheets. Система коментарів та пропозицій змін дозволяє ефективно координувати роботу команди над презентацією, забезпечуючи прозорий процес рецензування та вдосконалення матеріалів.

Gamma – сучасна платформа для створення інтерактивних презентацій, яка дозволяє легко структурувати матеріал, додавати інтегровані діаграми, UML-схеми чи візуалізації, роблячи захист більш динамічним і наочним.

Унікальною особливістю Gamma є використання штучного інтелекту для автоматичної генерації контенту та дизайну на основі введених тез, що значно прискорює процес створення професійних презентацій (рис. 6).

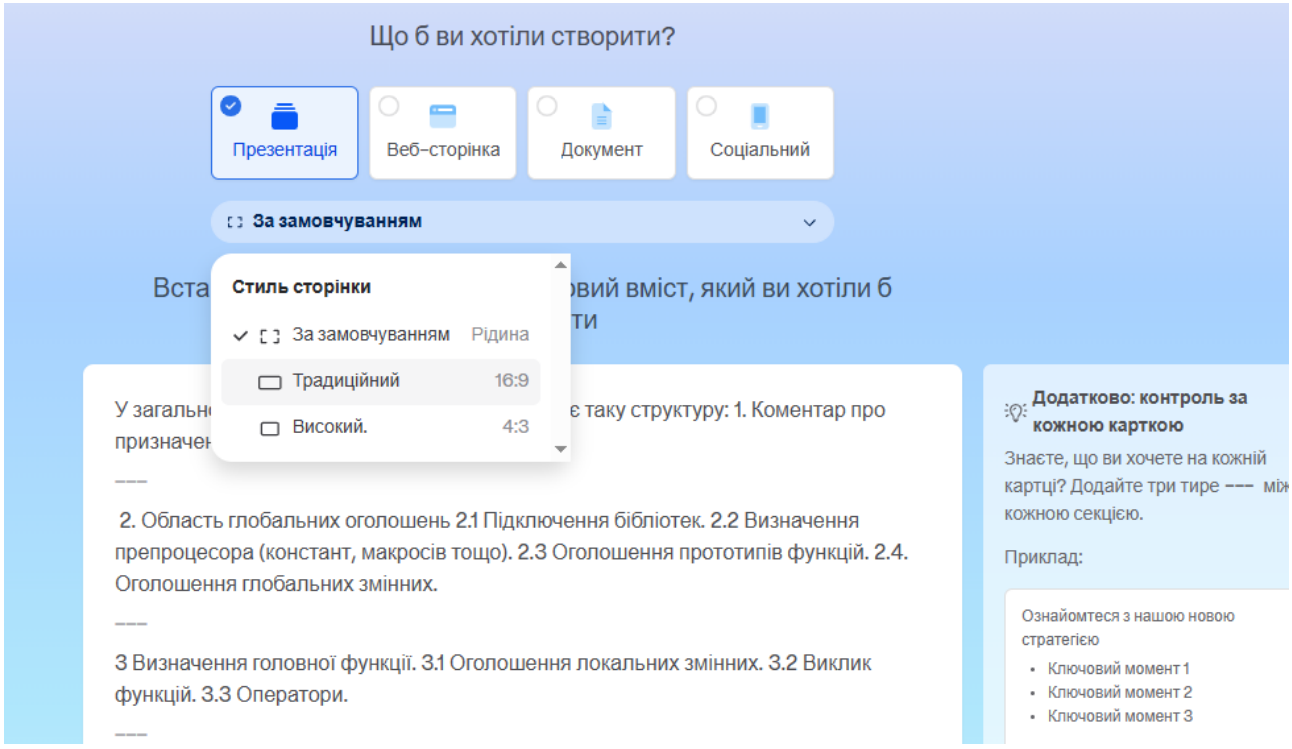


Рис. 6. Створення презентації у Gamma з використанням ШІ

Платформа підтримує адаптивний дизайн, який автоматично оптимізується для різних пристроїв, та дозволяє створювати нелінійні презентації з інтерактивними елементами, анімацією та вбудованими медіафайлами для більш захоплюючого представлення результатів проєкту.

Використання таких сервісів дає можливість не лише професійно оформити результати командної роботи, а й підвищити рівень взаємодії між студентами під час підготовки до захисту проєкту.

ЗРАЗОК КОНТРОЛЬНОЇ РОБОТИ

Завдання для виконання контрольної роботи №2
з дисципліни «Об’єктно-орієнтоване програмування»
для студентів 2 курсу напряму підготовки F3 Комп’ютерні науки
(до 15 балів максимально)
Варіант 1

Тестові завдання

2. Виберіть правильне оголошення похідного класу (0, 5 балів)

   А) class MoreDetails:: Details;

   Б) class MoreDetails: public Details;

   В) class MoreDetails: class(Details);

   Г) class MoreDetails: public class Details;

3. Яке ключове слово використовується при створення шаблонів класів чи функцій? (0, 5 балів)

   А) template

   Б) template_class

   В) temp

   Г) temp_class

4. Анонімну функцію всередині іншої функції можна визначити за допомогою ... (0, 5 балів)

   А) шаблону

   Б) лямбда-виразу

   В) інстанціювання шаблону

   Г) контейнера

5. Клас "circle.h" має поля radius та centrum та три віртуальних методи. Яка кількість VMT (таблиць віртуальних методів) створено для класу? (0, 5 балів)

   А) 1 таблиця

Б) 3 таблиці

В) 5 таблиць

Г) 2 таблиці

Ґ) жодної таблиці

6. Чи тотожні поняття "абстрактний клас" та "інтерфейс"? (0, 5 балів)

А) так, вони мають спільне призначення

Б) так, вони є абстрактними

В) ні, вони не тотожні

7. Що таке делегати? (0, 5 балів)

А) Це вказівники на методи, за допомогою яких можна викликати ці методи

Б) Це реалізація патерну Command (інкапсулювання у вигляді об'єкта будь-якого запиту/ виконання команди)

В) Це методи, які можна викликати безпосередньо з командного рядка

7. Навіщо потрібні узагальнення (Generic)? (0, 5 балів)

А) змінні типи Generic можуть містити одночасно кілька значень одного типу. Щоб отримати ці значення, використовується метод at()

Б) Generic використовуються для узагальнення типу, тобто є «універсальним» типом

В) Generic -типи можна кастувати в загальний тип object, на відміну від деяких інших

9. Має бути змога замість базового типу використати будь-який його підтип -- це принцип ... (0, 5 балів)

А) ідентифікації нащадків

Б) інверсії залежностей

В) єдиної відповідальності (клас має одну мету)

Г) підстановки

10. Базовим класом для всіх потоків є абстрактний клас … (0, 5 балів)

    А) TextReader

    Б) StreamWriter

    В) Stream

    Г) CanRead

11. Скільки конструкторів ініціалізації (з параметрами) може мати клас з двома полями? (0, 5 балів)

    А) жодного

    Б) один

    В) два

    Г) три

    Ґ) чотири

    Д) безліч

12. Якщо у похідному класі реалізується метод абстрактного класу, то при його оголошенні вказується ключове слово (0, 5 балів)

    А) abstract

    Б) override

    В) virtual

    Г) return

13. Для заборони успадкування з деякого класу або заборони перевизначення методу використовують ключове слово (0, 5 балів)

    А) base

    Б) abstract

    В) sealed

    Г) virtual

    Ґ) private

Практичні завдання

1. Пояснити, яким буде результат виконання коду (2 бали)

```
Person tom = new Employee();
tom.Print();  class Person {
public virtual void Print()
 {
      Console.Write("A1");
 } }
class Employee : Person {
public override void Print()
 {
      Console.Write("A2");
 }
}
```

2. Запропонувати зміни в архітектурі програми, так щоб спостерігалась відповідність базовим принципам SOLID (2 бали)

```
public class EmployeeReport
{
    /// <summary>
    /// Тип звіту
    /// </summary>
    public string TypeReport { get; set; }
    /// <summary>
    /// Звіт по працівнику
    /// </summary>
    public void GenerateReport(Employee em)
    {
        if (TypeReport == "CSV")
        {
            // Генерація звіту в форматі CSV
        }
        if (TypeReport == "PDF")
        {
            // Генерація звіту в форматі PDF
        }
    }
     }
```

3. Засобами мови C# створити клас «Traveler», клас «Route», клас «Trip», що містить дані про маршрут і мандрівника. Використати композицію , реалізувати методи

- AddTrip (Trip s) – додати подорож;
- CalculateDuration() – підрахунок тривалості подорожі
- ShowTripDetails() – вивід даних про мандрівку

Використовувати геттери-сеттери, інкапсуляцію, конструктори. (5 балів)

ПЕРЕЛІК ПИТАНЬ, ЩО ВИНОСЯТЬСЯ НА ЕКЗАМЕН

1 СЕМЕСТР

- Основні принципи та призначення об’єктно-орієнтованого програмування. Переваги над процедурним підходом.
- Клас. Структура класу на C++. Поняття об’єкта.
- Інкапсуляція, успадкування та поліморфізм в ООП.
- Специфікатори доступу у класах C++ (public, private, protected). Їх призначення та особливості.
- Поля та методи класу. Відмінність змінних-членів від локальних змінних.
- Використання гетерів і сетерів у класах C++. Призначення та особливості.
- Дружні функції у C++. В яких випадках доцільно їх застосовувати?
- Передача об’єктів класу як параметрів у методи. Особливості реалізації.
- Конструктори класів у C++. Призначення, види та правила оголошення.
- Конструктор копіювання. Його роль і відмінність від звичайного конструктора.
- Деструктор у C++. Призначення та правила використання.
- Поняття перевантаження операторів у C++. Які оператори можна перевантажувати?
- Приклади реалізації перевантаження арифметичних операторів. Особливості синтаксису.
- Перевантаження операторів введення та виведення (<< і >>) у класах.
- Вкладені класи. Сфера застосування та приклади використання.
- Композиція та агрегація класів. Відмінності та приклади реалізації.
- Використання композиції та агрегації у C++.
- Успадкування у C++. Види успадкування та їхні особливості.
- Відкрите успадкування. Його призначення та застосування.

- Просте та множинне успадкування у C++. Відмінності та проблеми реалізації.
- Віртуальні функції у C++. Призначення та приклади застосування.Таблиця віртуальних функцій (vtable).
- Абстрактні класи. Їх призначення та приклади використання у C++.
- Особливості визначення чисто віртуальних функцій у класах.
- Проблема діамантового успадкування. Шляхи розв’язання у C++.
- Використання ключового слова virtual при множинному успадкуванні.
- Застосування поліморфізму у C++. Динамічний поліморфізм. Відмінність між статичним і динамічним поліморфізмом у C++.
- Використання абстрактних класів у побудові ієрархії об’єктів.
- Поняття шаблонів (templates) у C++. Види шаблонів та їх призначення.
- Використання стандартних контейнерів STL (vector, list, map). Переваги над звичайними масивами.
- Опрацювання виняткових ситуацій у C++. Конструкції try, throw, catch.
- Стандартні винятки у C++ (std::exception та його нащадки). Приклади використання.

2 СЕМЕСТР

- .NET Framework. Зв’язок C# із платформою .NET?
- Роль відіграють CLI, CTS і CLS у забезпеченні роботи програми на C#
- Простори імен (namespaces) у C#, їх призначення
- Перетворення типів у C#.
- Умовні оператори та цикли C#. Оператори переходу (break, continue, return, goto).
- Масиви у C# . клас Array Різниця між одновимірними, багатовимірними та зубчастими масивами.

- Структура (struct) у C# . Відмінності від класу
- Класи String і StringBuilder. Методи для роботи з рядками (Substring, IndexOf, Replace, Split, Trim).
- Регулярні вирази. Методи класу Regex.
- Базовий клас Object у C#. Основні його методи
- Створення об’єктів класу у C#. Конструктори та деструктори.
- Модифікатори доступу у C#. Вплив на видимість членів класу.
- Індексатори у C#. Особливості реалізації.
- Властивості (properties) у C#. Аксесори get і set.
- Статичні члени класу.
- Перевантаження методів у C# та перевантаження операторів.
- Успадкування у C# . Його роль у повторному використанні коду.
- Віртуальні та абстрактні методи.
- Інтерфейс у C#, особливості реалізації.
- Делегати у C#, використання для виклику методів.
- Лямбда-вирази та узагальнення (generics) у C#. Приклади їх використання.
- Стандартні класи винятків у C# (Exception, IOException, ArgumentException та інші).
- Потоки. Робота з потоками вводу/виводу (класи Stream, StreamReader, StreamWriter).

ЗРАЗОК ЕКЗАМЕНАЦІЙНОГО БІЛЕТУ

Дрогобицький державний педагогічний університет імені Івана Франка
факультет фізики, математики, економіки та інноваційних технологій
кафедра фізики та інформаційних систем

ЕКЗАМЕНАЦІЙНИЙ БІЛЕТ №1

з дисципліни «Об'єктно-орієнтоване програмування»

Теоретичні питання

1. Основні поняття ООП. Клас. Структура класу. Екземпляри класу.
2. Поля та методи класу при успадкуванні. Директиви доступу.

Практичне завдання

Створити класи «Dish», «Menu», «Order». Замовлення містить список страв з меню. Використати композицію , реалізувати методи AddDish(Dish d) – додати страву та CalculateTotalCalories() – загальна калорійність. Використовувати геттери-сеттери, інкапсуляцію, конструктори.

Розглянуто на засіданні кафедри фізики та ІС від ___ _____ 20__ (протокол №__).

Викладач доц. Шаклеіна І.О.

Завідувач кафедри фізики та ІС доц. Гольський В.Б.

ГЛОСАРІЙ ТЕРМІНІВ

Абстракція – процес виділення суттєвих характеристик об’єкта без деталізації його реалізації.

Абстрактний клас – клас, який може містити як реалізовані методи, так і абстрактні, але не дозволяє створювати об’єкти напряму.

Абстрактний метод – метод без реалізації, який обов’язково має бути перевизначений у похідних класах.

Агрегація – слабкіший варіант композиції, коли об’єкт одного класу «посилається» на інший об’єкт, але не керує його життєвим циклом.

Асинхронне програмування – підхід, що дозволяє виконувати кілька операцій паралельно без блокування основного потоку.

Базовий клас – клас, від якого успадковуються інші класи.

Виключна ситуація (exception) – механізм обробки помилок у програмі, що дозволяє відокремити основний код від коду обробки помилок.

Віртуальна функція – метод у C++, який можна перевизначити в похідному класі; виклик відбувається динамічно завдяки механізму пізнього зв’язування.

Віртуальний метод – метод, який може бути перевизначений у похідному класі для реалізації поліморфізму.

Властивість (property) – член класу в C#, який інкапсулює поле та надає контрольований доступ до нього через get і set.

Гілка (branch) – паралельна версія коду в Git, яка дозволяє працювати над новими функціями незалежно від основної гілки.

Generics – аналог шаблонів у C#, який забезпечує створення типобезпечного узагальненого коду.

Git – система контролю версій, що дозволяє відстежувати зміни у проєкті та організовувати командну роботу.

CTS (Common Type System) – система загальних типів у .NET, яка забезпечує сумісність типів між різними мовами.

CLS (Common Language Specification) – набір правил, які визначають мінімальний функціонал для сумісності мов у межах .NET.

CLR (Common Language Runtime) – середовище виконання .NET, яке відповідає за управління пам'яттю, безпеку та виконання коду.

Делегат – тип у C#, що представляє посилання на метод; використовується для виклику методів через змінні.

Dependency Injection (DI) – спосіб реалізації IoC, коли залежності класів передаються через конструктор або властивості.

Деструктор – метод, який викликається при знищенні об'єкта для звільнення ресурсів.

Дружня функція (friend function) – функція, що не є членом класу, але має доступ до його приватних і захищених полів. Використовується для підвищення гнучкості взаємодії між класами.

Docker – система контейнеризації, яка дозволяє створювати, розгортати й запускати застосунки в ізольованих середовищах.

Життєвий цикл об'єкта – процес створення, використання та знищення екземпляра класу (визначається конструкторами та деструкторами).

Індексатор – спеціальний механізм у C#, який дозволяє об'єкту поводитися як масив.

Інкапсуляція – приховування внутрішньої реалізації об'єкта та надання доступу лише через публічні методи чи властивості.

Інверсія керування (IoC) – принцип, за яким об'єкт отримує свої залежності ззовні, а не створює їх самостійно.

Інтерфейс – абстрактний тип, який визначає набір методів і властивостей без реалізації.

Ітератор – об'єкт, який дозволяє послідовно перебирати елементи контейнера

Ініціалізатор – механізм встановлення початкових значень для полів об’єкта під час його створення.

Клас – основна структурна одиниця в ООП, що описує абстрактний тип даних із полями та методами. Клас є шаблоном для створення об’єктів.

Композиція – тип відношення між класами, коли один клас включає екземпляри інших класів як свої складові частини.

Конструктор – спеціальний метод класу, який автоматично викликається при створенні об’єкта і відповідає за його ініціалізацію.

Контейнер – структура даних, що зберігає набір об’єктів (vector, list, map у C++).

Лямбда-вираз – компактна форма оголошення анонімного методу у C#, яка дозволяє передавати функції як аргументи.

Метод – функція, визначена всередині класу, яка описує поведінку об’єкта.

Метод доступу (accessor) – спеціальний метод (get або set), що використовується для читання чи зміни значень приватних полів класу.

Merge – процес об’єднання змін із різних гілок у Git.

Мікросервіс – архітектурний стиль, при якому система складається з набору невеликих незалежних сервісів, які взаємодіють через API.

Множинне успадкування – можливість класу успадковувати властивості й методи від кількох базових класів (C++ підтримує, C# – ні).

Модифікатори доступу – ключові слова, що визначають рівень доступу до членів класу (public, private, protected).

Модифікатор sealed (C#) – модифікатор, що забороняє успадкування від класу.

Модуль – логічна частина програми, яка об’єднує пов’язані класи, функції чи інтерфейси для полегшення розробки та повторного використання коду.

Модульність – властивість програмної системи, за якої вона поділяється на незалежні частини (модулі), що спрощує супровід, тестування та повторне використання.

Об’єкт – екземпляр класу, який має власні значення полів і може виконувати методи, визначені у класі.

override – ключове слово в C#, яке використовується для перевизначення віртуального або абстрактного методу.

Пакет (package/assembly) – спосіб організації та поширення коду, що об’єднує класи, інтерфейси та ресурси в єдине ціле.

Патерн проєктування – готове архітектурне рішення для частої задачі у програмуванні, наприклад Singleton, Factory, Observer.

Перевантаження конструктора – можливість оголошення кількох конструкторів у класі з різними параметрами.

Перевантаження методів – визначення кількох методів з однаковим ім’ям, але різними параметрами.

Перевантаження операторів – визначення нової поведінки для стандартних операторів у користувацьких класах.

Перевантаження функцій – визначення кількох функцій із однаковим іменем, але з різними списками параметрів.

Перевизначення методів – зміна реалізації методу у похідному класі, що вже був визначений у базовому класі.

Поле класу – змінна, що належить класу і зберігає стан об’єкта.

Поліморфізм – здатність методів мати різну реалізацію залежно від типу об’єкта або параметрів.

Похідний клас – клас, який успадковує властивості та методи від базового класу.

Простір імен (Namespace) – механізм групування класів у C++ та C#, що дозволяє групувати класи та уникати конфліктів назв і уникнення конфліктів імен у великих проєктах.

Рефакторинг – процес зміни внутрішньої структури коду без зміни його зовнішньої поведінки.

SOLID-принципи – набір правил проектування об'єктно-орієнтованих систем, спрямованих на зручність підтримки та масштабованість.

STL (Standard Template Library) – бібліотека шаблонів у C++, яка містить контейнери, алгоритми та ітератори.

Статичне поле – змінна, що належить класу в цілому і є спільною для всіх його об'єктів.

Статичний метод – метод, що належить класу, а не конкретному об'єкту, і може викликатися без створення екземпляра.

Thread – потік виконання в програмі, який може працювати незалежно від інших.

try-catch-finally – конструкція для перехоплення та оброблення винятків у C++ та C#.

Unit-тестування – методика тестування, при якій перевіряються окремі модулі програми (класи, функції) ізольовано.

Успадкування – механізм ООП, який дозволяє створювати нові класи на основі вже існуючих.

Шаблон (template) – механізм у C++, що дозволяє створювати узагальнені функції або класи для роботи з різними типами даних.

Чисто віртуальна функція – метод у C++, що не має реалізації в базовому класі і робить клас абстрактним.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Грицюк Ю.І., Рак Т.Є. Об'єктно-орієнтоване програмування мовою С++ : навчальний посібник. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 404 с.
2. Григорович В.Г. Об'єктно-орієнтоване програмування. Частина 1. Магнолія, 2023. – 284с.
3. Григорович В.Г. Об’єктно-орієнтоване програмування. Успадкування. Навчальний посібник для студентів спеціальності «Комп’ютерні науки» . – Дрогобич: ДДПУ, 2018. – 219 с.
4. Григорович В.Г. Об’єктно-орієнтоване програмування. Віртуальні функції. Поліморфізм. Навчальний посібник для студентів спеціальності «Комп’ютерні науки» . – Дрогобич: ДДПУ, 2018. – 106 с.
5. Бублик В.В. Об’єктно-орієнтоване програмування: Підручник. – К. : ІТ книга, 2015. – 624с
6. Омельчук Л.Л. Об’єктно-орієнтоване програмування. Лабораторний практикум: навчальний посібник / Л.Л.Омельчук. – Київ: 2021. - 265 с.
7. Кравець П.О. Об'єктно-орієнтоване програмування. – Видавництво Львівської політехніки. 2012. – 624 с.
8. Решевська К. С., Лісняк А. О., Борю С. Ю. Об'єктно-орієнтоване програмування : навчальний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності "Комп'ютерні науки" освітньо-професійної програми "Комп'ютерні науки". Запоріжжя : ЗНУ, 2020. 94 с.
9. Щербаков О. В., Парфьонов Ю. Е., Федорченко В. М. Основи об'єктно-орієнтованого програмування: навчальний посібник. – Харків : ХНЕУ ім. С. Кузнеця, 2019. – 237 с.

10. Жуковський С.С., Вакалюк Т.А. Об‘єктно-орінтоване програмування мовою С++. Навчально-методичний посібник для студентів. – Житомир: Вид-во ЖДУ, 2016. – 100 с.
11. Львов М.С., Співаковський О.В. Вступ до об’єктно-орієнтоване програмування. Навчальний посібник. – Херсон: ХГПУ, 2000. – 238 с
12. Настенко Д.В., Нестерко А.Б. Об’єктно-орієнтоване програмування. Частина 1. Основи об’єктноорієнтованого програмування на мові С#.: Навчальний посібник. / Д.В. Настенко, А. Б. Нестерко. – К. : НТУУ «КПІ», 2016. – 76с.
13. Федорченко В.Ф., Лосєв М.Ю., Щербаков А.В., Парфенов Ю. Е. Об’єктно-орієнтоване програмування. Методичні рекомендації до виконання лабораторних робіт. Частина 2. – Харків, вид. ХНЕУ, 2009. – 72с.
14. Карнаух Т. О. Вступ до програмування мовою С++. Організація даних / Т. О. Карнаух, Ю. В. Коваль, М. В. Потієнко, А. Б. Ставровський. – К.: ВПЦ "Київський університет", 2015. – 215 с.
15. Ковалюк Т.В. Основи програмування. / Ковалюк Т.В. − Київ: BHV Київ, 2005. − 400с
16. Технології створення програмних продуктів та інформаційних систем : навч. посібник / М. Ю. Карпенко, Н. О. Манакова, І. О. Гавриленко ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2017. – 93 с
17. Інженерія якості програмного забезпечення: навч. посібник / Г.В Табунщик, Р.К. Кудерметов, Т.І. Брагіна. – Запоріжжя: ЗНТУ, 2013. – 180 с.

Навчальне видання

Ірина ШАКЛЕІНА, Андрій ПОПОВИЧ

Об’єктно-орієнтоване програмування:

методичні матеріали до самостійної роботи

навчальний посібник

Видавничий відділ
Дрогобицького державного педагогічного університету
імені Івана Франка

Комп’ютерне оформлення
Ірина Шаклеіна

Комп'ютерна верстка
Ірина Шаклеіна
Андрій Попович

Здано до набору 2. 11. 2025 р. Підписано до друку 22. 11. 2025 р.
Формат 60х90/16. Папір офсетний. Гарнітура. Times. Наклад 200 прим.

Ум. друк. арк. 6,25. Зам. № 70

Видавничий відділ Дрогобицького державного педагогічного університету імені Івана Франка
(Свідоцтво про внесення суб’єкта видавничої справи до державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції ДК № 2155 від 12. 04. 2005 р.)
82100 Дрогобич, вул. І.Франка, 24, к.42, тел. 2 – 23 – 78.